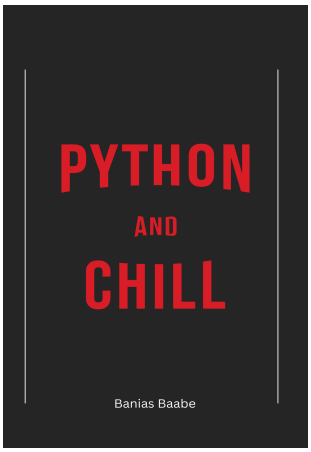
Python & Chill

Contents



Skip to main content

Why this Book exists

It was December 2022 when I started writing daily LinkedIn posts sharing one short Python/Data Science tip per day. My main goal was to share what I learnt over the years (and what I am currently learning) with others. I thought “When it’s useful for me, it will be probably useful for other people too”.

As time went by, more and more people found the tips useful. Soon, I realized that these daily tips could be compiled into a comprehensive guide for anyone interested in becoming better Data Scientist.

Thus, the idea for “Python & Chill” was born. The book contains all the tips shared by the author on LinkedIn, as well as many more to come in the future.

1. Code Quality

1.1. Automation

1.1.1. Automate Bash Commands with `Makefile`

Do you struggle to remember the exact series of commands needed to build and package a Python project?

Or manually running tests and installing dependencies?

`Makefile` to the rescue!

A `Makefile` can be used to automate bash commands and have a standardized way to execute those.

See below for how we define commands for testing, install our dependencies, and format our code with black.

```
# Create a file named Makefile
# Makefile

install:
    @echo "Installing requirements ..."
    pip install -r requirements.txt --quiet

test:
    @echo "Testing ..."
    pytest

format:
    @echo "Formatting ..."
    black .
```

1.1.2. Automate Testing with Nox

Testing your code against multiple Python Versions is hard.

With `Nox`, you can automate this step!

`Nox` is a command-line tool to automate testing in multiple Python Environments locally.

You can customize the sessions with a Python script.

See the example below, where we define a session (**tests**) in our **noxfile.py** and define the Python versions we want to test against. Moreover we set up another session (**lint**) to run flake8 against our code.

`Nox` is highly customizable, so check out their documentation.

```
# noxfile.py

import nox

@nox.session(python=["3.6", "3.7", "3.8", "3.9"])
def tests(session):
    # Install testing dependency
    session.install('pytest')
    # Run tests
    session.run('pytest')

@nox.session
def lint(session):
    session.install("flake8")
    session.run("flake8", "example.py")
```

```
!pip install nox
```

```
!nox
```

1.1.3. Update Your Dependencies Automatically with `pyup`

Do you need an easy way to update your project's dependencies?

Try `pyup`.

`pyup` goes through your dependency files and searches for new package versions.

It will then create a new branch in your repository, a new commit for every update and a pull request containing all commits.

You only need to provide an access token (from GitHub or GitLab) and the repository name.

```
!pip install pyupio
```

```
!pyup --repo=username/repo --user-token=<YOUR_TOKEN> --initial # After running the
```

Skip to main content

```
# gitlab.com:
!pyup --provider gitlab --repo=username/repo --user-token=<YOUR_TOKEN>
```

1.2. CI/CD

1.2.1. Test GitHub Actions Workflows Locally with `act`

Don’t waste time when testing Github Actions.

Instead, use `act`.

`act` allows you to run your Github Actions workflows locally.

Instead of committing/pushing every time you make changes to your Workflow files.

It uses Docker to pull or build the necessary images, as defined in your workflow files. So you need Docker installed on your machine.

```
!curl -s https://raw.githubusercontent.com/nektos/act/master/install.sh | sudo bash
```

```
act
```

1.2.2. Cache Python Dependencies in GitHub Actions

Don’t waste time for your GitHub Actions workflows.

Whenever a new run is triggered, your dependencies are probably installed again and again and again…

Even if you didn’t change anything in your dependency file.

With the small snippet below, the action caches your dependencies so you can skip the install step the next time your workflow runs.

Skip to main content

```
name: ci

on:
  push:
    branches: [ main ]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v4
      with:
        python-version: '3.9'
        cache: 'pip' # caching pip dependencies (supports poetry and pipenv too)
    - run: pip install -r requirements.txt

    - name: Run Python unit tests
      run: python3 -u -m unittest tests/tests.py
```

1.2.3. Lint Github Actions Workflow Files with `actionlint`

To lint your Github actions workflow files, use `actionlint`.

`actionlint` is a tool for detecting potential problems and smells in your workflow files.

This includes

- Syntax checks
- Security checks
- Cron syntax checking
- Checks for access to non-existent properties

It has a pre-commit hook too.

```
!go install github.com/rhysd/actionlint/cmd/actionlint@latest
```

```
!actionlint
```

1.3. Code Style

1.3.1. Remove unused lines with `autoflake`

Do you always remove your unused imports and variables in Python?

If not, this can be problematic:

- Your code will be more difficult to read
- Reduced performance because the Python interpreter has to spend less time importing and evaluating unnecessary code
- Potential issues can arise such as conflicts with other variables

If you want to clean and optimize your code, try `autoflake`

`autoflake` is a tool that automatically removes unused imports and variables for you.

You can also customize to fine-tune how `autoflake` processes your code.

```
!pip install autoflake
```

```
!autoflake --in-place <example.py>
```

1.3.2. Ensure Documentation with `interrogate`

One aspect of code quality and maintainability is documenting your code.

In Python, there are docstrings to do this job.

How about to check if all of your methods and classes use docstrings?

Try `interrogate`.

Skip to main content

You can even print out a coverage report.

Say goodbye to poorly-documented code!

```
!pip install interrogate
```

```
!interrogate . -v
```

1.3.3. Sort your imports automatically with `isort`

When your Python app gets bigger, you will have more and more import statements.

This can look nasty when you don't order them.

Don't sort them manually.

Instead, use `isort`.

`isort` is a Python library that sorts your imports alphabetically and separates them into sections.

You only need to type one line into your CLI.

```
!pip install isort
```

```
!isort .
```

1.3.4. Format your code automatically with `black`

If you want to avoid manual code formatting, use `black` for Python.

`black` is a library providing automatic formatting of Python code.

It makes it easier for you to focus on writing high-quality code.

Skip to main content

```
!pip install black
```

```
!black .
```

1.3.5. Lint your YAML files with `yamllint`

Are you tired of manually checking YAML files for errors and formatting issues?

Use `yamllint`.

`yamllint` is a Python library to lint YAML files, ensuring that they are well-formed.

Since yaml is a popular choice for Config files, yamllint is a nice tool to catch errors before they cause problems.

```
!pip install yamllint
```

```
!yamllint .
```

1.3.6. Blazingly fast linting with `ruff`

Are you still using flake8 or pylint?

Instead, use `ruff`, with 10x-100x faster performance.

`ruff` is a blazingly fast linter for Python, written in Rust.

It is orders of magnitudes faster than flake8 and pylint while integrating more functionality like import sorting or removing unused code.

```python
# some_file.py
import requests
import json

def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError as e:
        result = 'infinity'
    return result
```

```
!pip install ruff
```

```
!ruff some_file.py
```

1.3.7. Format Python Code in Documentation Files

Do you want to apply black on your documentation files?

Try `blacken-docs`!

`blacken-docs` apply the popular code formatting tool black in your documentation.

It supports Code snippets in:

- Markdown
- LaTeX
- reStructuredText

Don't write unformatted code in your documentation anymore!

```
!pip install blacken-docs
```

```
!blacken-docs TEST.md
```

Skip to main content

1.3.8. Correct Misspellings in Your Codebase with `codespell`

Typos in your codebase are nasty.

With `codespell`, you can correct them automatically.

`codespell` corrects your misspelled words in your source code and other files.

But it makes sure to not touch niche terms to reduce false positives.

Clean up your codebase.

```
!pip install codespell
```

```
!codespell # Running codespell in all files of current directory
```

1.3.9. Lint Your Dockerfile with `hadolint`

Do you want to lint your Dockerfiles?

Try `hadolint`.

`hadolint` enforces Dockerfile best practices by parsing the Dockerfile into an AST and performing rules.

It's available as a Docker container image.

```
!docker run -rm -i hadolint/hadolint < Dockerfile
```

1.3.10. Detect Typos in your codebase with `typos`

To find typos in your code base, use `typos`.

Skip to main content

It has a low false positive rate, so you can easily run it in your CI pipeline or as a pre-commit hook.

```
!brew install typos-cli
```

```
!typos
```

1.4. Memory Optimization

1.4.1. Identify your bottleneck regarding memory with `memory_profiler`

Want to identify which lines use the most amount of memory in your Python script?

Try `memory_profiler`.

`memory_profiler` is a Python module to make a line-by-line analysis of memory consumption for Python functions.

Below you can see how to use `memory_profiler` within your Python script.

- Decorate the function you want to profile with `@profile`.
- Run the script by passing the option `-m memory_profiler` to load the `memory_profiler` module.

```
from memory_profiler import profile

@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

my_func()
```

```
!pip install memory_profiler
```

Skip to main content

```
!python -m memory_profiler memory.py
```

```
'''
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3     41.9 MiB     41.9 MiB           1   @profile
     4                                         def my_func():
     5     49.5 MiB      7.6 MiB         102       a = [i**2 for i in range(1,100)]
     6    194.5 MiB    145.0 MiB          22       b = [i**i for i in range(1,20)]
     7    194.5 MiB      0.0 MiB           1       return a, b
'''
```

1.5. Security in Projects

1.5.1. Detect Common Security Issues with `bandit`

Do you want to find potential security issues in your Python code?

Try using `bandit`.

`bandit` is a Python package to find common security issues and known vulnerabilities automatically.

It works by processing files to create an abstract syntax tree, which is then used to run plugins against. It then produces a report on the results.

In the example below, we will try to use the `requests` library and ignore verifying the SSL certificate with **verify=False**.

`bandit` will immediately identify this line as a security issue.

```
# bandit_test.py
import requests

data = requests.get("https://www.google.de/", verify=False)
```

```
!pip install bandit
```

Skip to main content

```
!bandit -r bandit_test.py
```

```
'''
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.10.8
[node_visitor]  WARNING Unable to find qualified name for module: bandit_test.py
Run started:2022-12-23 15:32:44.650893
Test results:
>> Issue: [B501:request_with_no_cert_validation] Requests call with verify=False dis
   Severity: High   Confidence: High
   CWE: CWE-295 (https://cwe.mitre.org/data/definitions/295.html)
   Location: bandit_test.py:3:7
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b501_request_with_no_ce
2
3       data = requests.get("https://www.google.de/", verify = False)
4       print(data.status_code)
--------------------------------------------------
Code scanned:
        Total lines of code: 3
        Total lines skipped (#nosec): 0
Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 1
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 1
Files skipped (0):
'''
```

1.5.2. Detect vulnerabilities in your Environment

Do you want to detect vulnerabilities in your Python environment?

Try `pip-audit`.

`pip-audit` is a CLI tool to detect vulnerabilities in the packages installed in your Python environment

Skip to main content

It checks your packages against the Python Packaging Advisory Database.

The tool also provides suggestions to which version you should upgrade your package.

```
!pip install pip-audit
```

```
!pip-audit
```

```
"""
Found 3 known vulnerabilities in 2 packages
Name       Version ID                  Fix Versions
---------- ------- ------------------- ------------
flask      0.5     PYSEC-2019-179      1.0
flask      0.5     PYSEC-2018-66       0.12.3
setuptools 56.0.0  GHSA-r9hx-vwmv-q579 65.5.1
"""
```

1.5.3. Store Credentials safely with `keyring`

Almost every application needs credentials like password or API keys.

But you should never store them in plain text files. They would be trivially accessible to anybody who has access to the text file.

To store credentials securely, use `keyring`.

`keyring` provides a Python wrapper around your system's password store (macOS Keychain, Windows Credential Locker, etc.), which is safer than a plain text file.

The example below stores and retrieves the password easily, but you can store any of the other fields.

(This can also be done through CLI, since `keyring` also comes with a command-line functionality).

```
!pip install keyring
```

```
import keyring

# set your password
keyring.set_password("mydb", "username", "password")

# get your password
keyring.get_password("mydb", "username")
```

1.6. Typing

1.6.1. Enforce types with `typeguard`

How to enforce types in Python easily?

You can try type hinting and define the expected types.

But you can still pass the wrong types and get away without an error.

Instead, use `typeguard`.

`typeguard`'s decorator makes sure that an error gets raised when a variable of a wrong type is passed.

See below, how you only need to add a decorator to the function to make things work.

```
!pip install typeguard
```

```
from typeguard import typechecked

@typechecked
def say_hello(name: str):
    print(f"Hello, {name}")


say_hello(3)
```

1.6.2. Static type checking with `mypy`

Do you want to catch type errors before they cause problems in production?

Try `mypy`!

`mypy` is a static type checker for Python that can help you catch type errors before you even run your code.

It analyzes your code to ensure that the types of your variables and functions align with the expected types.

So `mypy` helps you to write more reliable code.

```
# bank.py
def deposit(amount: int, balance: int):
        return balance + amount

def withdraw(amount: int, balance: int):
        return balance - amount


deposit(100, 1000)
withdraw(100, "1000")
```

```
!pip install mypy
```

```
!mypy bank.py
```

1.6.3. Faster Static Type Checking with `pyright`

Do you wish to find type errors before they make problems in production?

Use `pyright`!

`pyright` is a fast static type checker for Python, which helps you find type errors even before you

Skip to main content

It checks your code to make sure that the types of your variables and functions match what is expected.

`pyright` is written in TypeScript, so check the community version to avoid going through installing extra things (Link is below).

It's 3x-5x times than the OG static type checker, mypy.

Whether it's pyright or mypy, please consider to integrate static type checkers.

```
!pip install pyright
```

```
!pyright
```

1.7. Principles for Code Quality

1.7.1. The Law of Demeter

It's also known as the Principle of Least Knowledge, saying that an object should only communicate with its immediate neighbors, avoiding to access deeper and deeper objects.

See below for a small example how we would violate and obey the Law of Demeter.

```python
# Bad Example
class Department:
    def __init__(self, manager):
        self.manager = manager

    def get_manager_name(self):
        # Bad: Accessing a method of an object deep within the hierarchy
        return self.manager.employee.name

class Employee:
    def __init__(self, name):
        self.name = name

class Manager:
    def __init__(self, employee):
        self.employee = employee

# Good Example
class Department:
    def __init__(self, manager):
        self.manager = manager

    def get_manager_name(self):
        # Good: Not going deeper
        return self.manager.get_name()

class Employee:
    def __init__(self, name):
        self.name = name

class Manager:
    def __init__(self, employee):
        self.employee = employee

    def get_name(self):
        return self.employee.name
```

2. Cool Tools

2.1. Cool Tools

2.1.1. Work with Countries, Currencies, Subdivisions, and more

You probably know how important it is to use the correct codes for countries, currencies, languages, and subdivisions.

To save the headache, try `pycountry` for Python!

`pycountry` makes it easy to work with these codes.

It allows you to look up country and currency information by name or code based on ISO.

But it can also be used to get the name or code for a specific currency or country.

```
!pip install pycountry
```

```
import pycountry

# Get Country
print(pycountry.countries.get(alpha_2="DE"))

# Get Currency
print(pycountry.currencies.get(alpha_3="EUR"))

# Get Language
print(pycountry.languages.get(alpha_2='DE'))
```

2.1.2. Generate better requirements files with `pipreqs`

To generate a requirements.txt file, don't do pip freeze > requirements.txt

It will save all packages in your environment including those you are not currently using in your project (but still have installed).

Instead, use `pipreqs`.

`pipreqs` will only save those packages based on imports in your project.

A very good option for plain virtual environments.

```
!pip install pipreqs
```

Skip to main content

```
!pipreqs .
```

2.1.3. Remove a package and its dependencies with `pip-autoremove`

When you want to remove a package via pip, you will encounter following problem:

pip will remove the desired package but not its unused dependencies.

Instead, try `pip-autoremove`.

It will automatically remove a package and its unused dependencies.

A really good option when you are not using something like Poetry.

```
!pip install pip-autoremove
```

```
!pip-autoremove flask -y
```

2.1.4. Get distance between postal codes

Do you want the distance between two postal codes?

Use `pgeocode`.

Just specify your country + postal codes and get the distance in KM.

```
!pip install pgeocode
```

```
import pgeocode

dist = pgeocode.GeoDistance('DE')
dist.query_postal_code('10117', '80331')
```

Skip to main content

2.1.5. Working with units with `pint`

Have you ever struggled with units in Python?

With `pint`, you don't have to.

`pint` is a Python library for easy unit conversion and manipulation.

You can handle physical quantities with units, perform conversions, and perform arithmetic with physical quantities.

With `pint`, you keep track of your units and ensure accurate results.

```
!pip install pint
```

```
import pint

# Initializing the unit registry
ureg = pint.UnitRegistry()

# Defining a physical quantity with units
distance = 33.0 * ureg.kilometers
print(distance)
# 33.0 kilometer

# Converting between units
print(distance.to(ureg.feet))
# 108267.71653543308 foot

# Performing arithmetic operations
speed = 6 * distance / ureg.hour
print(speed)
# 198.0 kilometer / hour
```

2.1.6. Supercharge your Python profiling with `Scalene`

Want to identify Python performance issues?

Try `Scalene`, your Profiler on steroids!

Skip to main content

`Scalene` is a Python CPU + GPU + Memory profiler to identify bottlenecks.

Even with AI-powered optimization proposals!

`Scalene` comes with an easy-to-use CLI and web-based GUI.

```
!pip install scalene
```

```
!scalene <my_module.py>
```

2.1.7. Fix unicode errors with `ftfy`

Have you ever struggled with Unicode errors in your Python code?

Try `ftfy`!

`ftfy` repairs scrambled text which occurs as a result of encoding or decoding problems.

You will probably know it when text in a foreign language can’t appear correctly.

In Python you only have to call one method from `ftfy` to fix it.

```
!pip install ftfy
```

```
import ftfy

print(ftfy.fix_text('What does â€œftfyâ€\x9d mean?'))
print(ftfy.fix_text('â“ Check'))
print(ftfy.fix_text('The Mona Lisa doesnÃƒÂ¢Ã¢â€šÂ¬Ã¢â€žÂ¢t have eyebrows.'))
```

2.1.8. Remove the background from images with `rembg`

Do you want to remove the background from images with Python?

Use `rembg`.

Skip to main content

With its pre-trained models, `rembg` makes removing the background of your images easy.

```
!pip install rembg
```

```
from rembg import remove
import cv2

input_path = 'car.jpg'
output_path = 'car2.jpg'

input_file = cv2.imread(input_path)
output_file = remove(input_file)
cv2.imwrite(output_path, output_file)
```

2.1.9. Build modern CLI apps with `typer`

Tired of building clunky CLI for your Python applications?

Try `typer`.

`typer` makes it easy to create clean, intuitive CLI apps that are easy to use and maintain.

It also comes with auto-generated help messages.

Ditch argparse.

```
!pip install typer
```

```python
# hello_script.py
import typer

app = typer.Typer()

@app.command()
def hello(name: str):
    typer.echo(f"Hello, {name}!")

@app.command()
def bye(name: str):
    typer.echo(f"Bye, {name}!")

if __name__ == "__main__":
    app()
```

```
!python hello_script.py hello John
```

2.1.10. Generate realistic fake data with `faker`

Creating realistic test data for your Python projects is annoying.

`faker` helps you to do that!

With just a few lines of code, you can generate realistic and diverse test data, such as :

- Names
- Addresses
- Phone numbers
- Email addresses
- Jobs

And more!

You can even set the local or language for more diverse output.

```
!pip install faker
```

Skip to main content

```
from faker import Faker
fake = Faker('fr_FR')
print(fake.name())
print(fake.job())
print(fake.phone_number())
```

2.1.11. Enrich your progress bars with `rich`

Do you want a more colorful output for progress bars?

Use `rich`

`rich` offers a beautiful progress bar, instead of tqdm's boring output.

With `rich.progress.track`, you can get a colorful output.

```
!pip install rich
```

```
from rich.progress import track
for url in track(range(25000000)):
    # Do something
    pass
```

2.1.12. Set the description for `tqdm` bars

When you work with progress bars, you will probably use **tqdm**.

Do you know you can add descriptions to your bar?

You can do that with `set_description()`.

```
import tqdm
import glob

files = tqdm.tqdm(glob.glob("sample_data/*.csv"))
for file in files:
    files.set_description(f"Read {file}")
```

Skip to main content

2.1.13. Convert Emojis to Text with `emot`

Analyzing emojis and emoticons in texts can give you useful insights.

With `emot`, you can convert emoticons into words.

Especially useful for sentiment analysis.

```
!pip install emot
```

```
import emot
emot_obj = emot.core.emot()
text = "I love python ☮ ❤ :-) :-( :-)))"
emot_obj.emoji(text)
```

2.1.14. Print hardware information and version numbers

When raising an issue, you should provide version numbers and hardware information.

With `watermark`, you can do that easily.

Just install the package and print.

```
!pip install watermark
```

```
from watermark import watermark
print(watermark())
```

2.1.15. Cache requests with `requests-cache`

Do you want better performance for requests?

Use `requests-cache`.

Skip to main content

It caches HTTP requests so you don't have to make the same requests again and again.

In the example below, a test endpoint with a 1-second delay will be called.

With the standard `requests` library, this takes 60 seconds.

With `requests-cache`, this takes 1 second.

```
!pip install requests-cache
```

```
# This takes 60 seconds
import requests

session = requests.Session()
for i in range(60):
    session.get('https://httpbin.org/delay/1')



# This takes 1 second
import requests_cache

session = requests_cache.CachedSession('test_cache')
for i in range(60):
    session.get('https://httpbin.org/delay/1')
```

2.1.16. Unify messy columns with `unifyname`

Do you want to unify messy string columns?

Try `unifyname`, based on fuzzy string matching.

This small library cleans up your messy columns with 100s of different variations for one word.

```
!pip install unifyname
```

```
import pandas as pd
from unifyname.utils import unify_names, deduplicate_list_string

data = pd.read_csv("")

data["BAIRRO DO IMOVEL"].value_counts()

data = unify_names(data,column='BAIRRO DO IMOVEL',threshold_count=500)

data["BAIRRO DO IMOVEL"].value_counts()
```

2.1.17. Check for broken links in a website

`linkchecker` is a Python library for recursively going through a website and checking for broken links.

You may not have the time to do that manually.

And broken links can harm your Search Engine Ranking.

See below how easy it can be to set up and use.

```
!pip install linkchecker
```

```
!linkchecker https://www.example.com
```

2.1.18. Matplotlib for your Terminal

`bashplotlib` is a little library that displays basic ASCII graphs in your terminal.

It provides a quick way to visualize your data.

Currently, `bashplotlib` only supports histogram and scatter plots.

```
!pip install bashplotlib
```

Skip to main content

```
!hist --file test.txt
```

2.1.19. Display a Dependency Tree of your Environment

Do you want to stop resolving dependency issues?

Try `pipdeptree`.

`pipdeptree` displays your installed Python packages in the form of a dependency tree.

It will also show you warnings when there are possible version conflicts.

An alternative to tools like Poetry which resolves dependency issues for you automatically.

```
!pip install pipdeptree
```

```
!pipdeptree
```

2.1.20. Sort LaTeX acronyms automatically

I wrote a small library (`acrosort-tex`) to sort LaTeX acronyms with one command automatically.

It was a fun Sunday project where I really learned how easy it is to publish a package with Poetry.

Currently, it only supports acronyms in the following format:

\□□□□{□□□□□□□□□□□□□}[□□□□□□□□□□]{□□□□□□□□□}

but it’s a beginning :)

See below for a small example.

Link to the repository: https://lnkd.in/eTF8qs5w

```
!pip install acrosort_tex
```

Skip to main content

```
!acrosort old.tex new.tex
```

2.1.21. Make ASCII Art from Text

Create ASCII Art From Text in your Terminal

With `pyfiglet`, you can generate banner-like text with Python.

This is a nice feature to introduce your users to your Python CLI apps.

```
!pip install pyfiglet
```

```
# Default font
ascii_art = pyfiglet.figlet_format('Hello, world!')

# Alphabet font
ascii_art = pyfiglet.figlet_format('Hello, world!', font='Alphabet')

# Bubblehead font
ascii_art = pyfiglet.figlet_format('Hello, world!', font='bulbhead')
```

2.1.22. Display NER with `spacy`

If you want to perform and visualize Named-entity Recognition, use `spacy.displacy`.

It makes NER and visualizing detected entities super easy.

`displacy` has some other cool tools like visualizing dependencies within a sentence or visualizing spans, so check it out.

```
import spacy
from spacy import displacy

text = "Chelsea Football Club is an English professional football club based in Full
        Founded in 1905, they play their home games at Stamford Bridge. \
        The club competes in the Premier League, the top division of English footbal
        They won their first major honour, the League championship, in 1955."

nlp = spacy.load("en_core_web_sm")
doc = nlp(text)
displacy.render(doc, style="ent", jupyter=True)
```

2.1.23. Create TikZ pictures with Python

If you have ever written a paper in LaTeX, you probably used TikZ for your graphics.

TikZ is probably the most powerful tool to create graphic elements.

And notoriously hard to learn.

No need to worry, you can create TikZ-figures in Python too.

With `tikzplotlib`, you can convert matplotlib figures into TikZ.

You can then insert the resulting plot in your LaTeX file.

Really useful when you don't want the hassle with TikZ.

```
!pip install tikzplotlib
```

```
import tikzplotlib
import matplotlib.pyplot as plt
import numpy as np

plt.style.use("ggplot")

t = np.arange(0.0, 2.0, 0.1)
s = np.sin(2 * np.pi * t)
s2 = np.cos(2 * np.pi * t)
plt.plot(t, s, "o-", lw=4.1)
plt.plot(t, s2, "o-", lw=4.1)
plt.xlabel("time (s)")
plt.ylabel("Voltage (mV)")
plt.title("Simple plot $\\frac{\\alpha}{2}$")
plt.grid(True)

tikzplotlib.save("mytikz.tex")
```

2.1.24. Human-readable RegEx with `PRegEx`

RegEx is notoriously nasty to read and write.

For a human-readable alternative, try `PRegEx`.

`PRegEx` is a Python library aiming to have an easy-to-remember syntax to write RegEx patterns.

It offers a way to easily break down a complex pattern into multiple simpler ones that can then be combined.

See below how we can write a pattern that matches any URL that ends with either “.com” or “.org” as well as any IP address for which a 4-digit port number is specified.

```
from pregex.core.classes import AnyLetter, AnyDigit, AnyFrom
from pregex.core.quantifiers import Optional, AtLeastAtMost
from pregex.core.operators import Either
from pregex.core.groups import Capture
from pregex.core.pre import Pregex

http_protocol = Optional('http' + Optional('s') + '://')

www = Optional('www.')

alphanum = AnyLetter() | AnyDigit()

domain_name = \
  alphanum + \
  AtLeastAtMost(alphanum | AnyFrom('-', '.'), n=1, m=61) + \
  alphanum

tld = '.' + Either('com', 'org')

ip_octet = AnyDigit().at_least_at_most(n=1, m=3)

port_number = (AnyDigit() - '0') + 3 * AnyDigit()

# Combine sub-patterns together.
pre: Pregex = \
    http_protocol + \
    Either(
        www + Capture(domain_name) + tld,
        3 * (ip_octet + '.') + ip_octet + ':' + port_number
    )
```

2.1.25. Perform OCR with `easyOCR`

Effortlessly extract text from Images with `EasyOCR`

`EasyOCR` is a Python library for Optical Character Recognition (OCR), built on top of PyTorch.

It supports over 80 languages and writing scripts like Latin, Chinese, Arabic and Cyrillic.

See below how easy we can extract text from a given image.

PS: Even if it's working on CPU, running on GPU is recommended.

```
!pip install easyocr
```

```
import easyocr

reader = easyocr.Reader(['en'])
image_path = 'english_image.png'
results = reader.readtext(image_path)

for result in results:
    text = result[1]
    print(text)
```

2.1.26. Diagram-as-Code with `diagrams`

With the package `diagrams`, you can draw various types of diagrams with Python code.

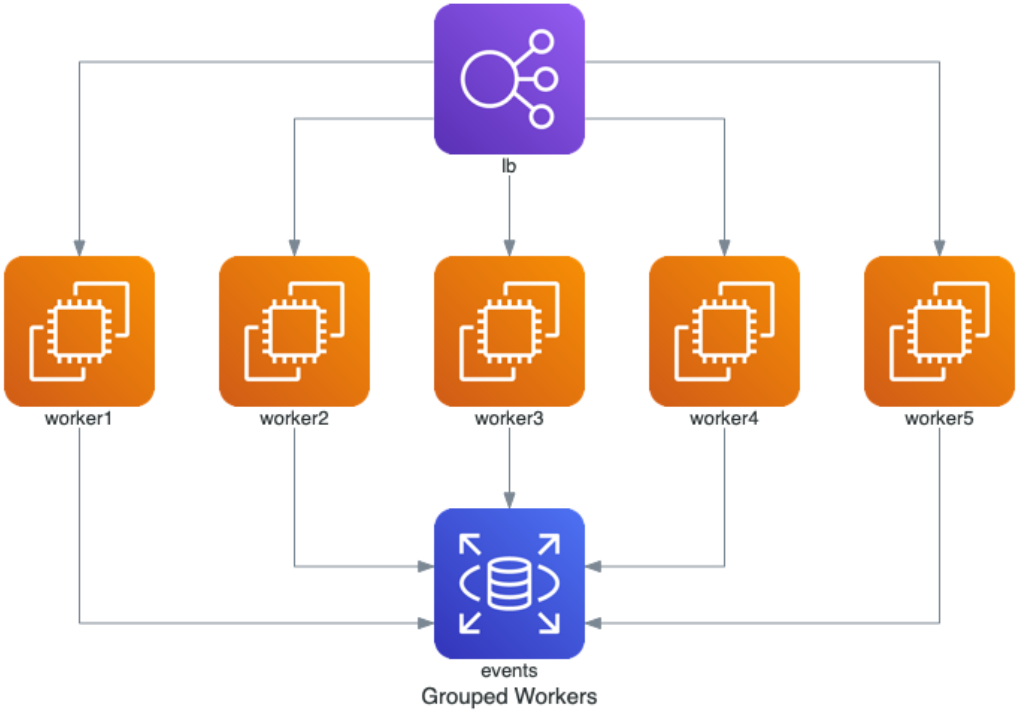
It offers a simple syntax and nodes from many Cloud Providers like AWS, Azure or GCP.

See below how easy it is to draw a simple architecture.

```
!pip install diagrams
```

```
from diagrams import Diagram
from diagrams.aws.compute import EC2
from diagrams.aws.database import RDS
from diagrams.aws.network import ELB

with Diagram("Grouped Workers", show=False, direction="TB"):
    ELB("lb") >> [EC2("worker1"),
                  EC2("worker2"),
                  EC2("worker3"),
                  EC2("worker4"),
                  EC2("worker5")] >> RDS("events")
```



2.1.27. Powerful Retry Functionality with `tenacity`

What, if an API call in your program fails?

Because of, let's say, instable internet connection?

This is not so uncommon.

You usually should have some sort of retry mechanism in your program.

With `tenacity` in Python, this isn't a problem anymore.

`tenacity` offers a retrying behaviour with a decorator with powerful features like:

- Define Stop Conditions
- Define Wait Conditions
- Customize retrying on Exception
- Retry on Coroutines

```
import tenacity as t

# Stop after N attempts
@retry(stop=t.stop_after_attempt(5))
def stop_after_5_attempts():
    print("Stopping after 5 attempts")
    raise Exception

# OR Condition
@retry(stop=(t.stop_after_delay(10) | t.stop_after_attempt(5)))
def stop_after_10_s_or_5_retries():
    print("Stopping after 10 seconds or 5 retries")
    raise Exception
# Wait for X Seconds
@retry(wait=t.wait_fixed(2))
def wait_2_s():
    print("Wait 2 second between retries")
    raise Exception

# Retry for specific Exceptions
@retry(retry=t.retry_if_exception_type(IOError))
def might_io_error():
    print("Retry forever with no wait if an IOError occurs, raise any other errors"
    raise Exception
```

2.1.28. Performant Graph Analysis with `python-igraph`

When you want to work with graphs in Python

Use `python-igraph`.

`python-igraph` offers a Python Interface to igraph, a fast and open source C library to manipulate and analyze graphs.

Due to its high performance, it can handle larger graphs for complex network research which you can visualize with matplotlib or plotly.

It's documentation also offers neat tutorials for different purposes.

```
!pip install igraph
```

```
import igraph as ig
import matplotlib.pyplot as plt

g = ig.Graph(
    6,
    [(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)]
)

g.es['width'] = 0.5

fig, ax = plt.subplots()
ig.plot(
    g,
    target=ax,
    layout='circle',
    vertex_color='steelblue',
    vertex_label=range(g.vcount()),
    edge_width=g.es['width'],
    edge_color='#666',
    edge_background='white'
)
plt.show()
```

2.1.29. Speedtests via CLI with `speedtest-cli`

If you want to test your internet bandwidth via your CLI

try `speedtest-cli`.

`speedtest-cli` tests your internet bandwidth via speedtest(dot)net.

It's installable via pip.

```
!pip install speedtest-cli
```

```
!speedtest-cli
```

2.1.30. Minimalistic Database for Python with `tinydb`

Do you search for a minimalistic document-oriented database in Python?

Skip to main content

Use `tinydb`.

`tinydb` is written in pure Python and offers a lightweight document-oriented database.

It's perfect for small apps and hobby projects.

```
!pip install tinydb
```

```
from tinydb import TinyDB, Query

db = TinyDB('/path/to/db.json')
db.insert({'int': 1, 'char': 'a'})
db.insert({'int': 1, 'char': 'b'})
```

2.1.31. Calculate Code Metrics with `radon`

How do you ensure your codebase stays clean and maintainable?

What if you can calculate how complex your codebase is?

There are different metrics to do that:

- Raw Metrics like Source Lines of Code (SLOC) or Logical Lines of Code (LLOC). They are not a good estimator for the complexity.
- Cyclomatic Complexity: Corresponds to the number of decisions in the code + 1 (e.g. every for or if counts).
- Halstead Metrics: Metrics derived from the number of distinct and total operators and operands.
- Maintainability Index: Measures how maintainable the code is. It's a mix of SLOC, Cyclomatic Complexity, and a Halstead Metric.

With `radon`, you can calculate those metrics described above in Python (or via CLI).

```
!pip install radon
```

```
!radon cc example.py
```

Skip to main content

2.1.32. Better Alternative to `requests`

Want a better alternative to `requests`?

Use `httpx` for Python.

`httpx` is a modern alternative to `requests` to make HTTP requests (while having a similar API).

One of the main advantages is it supports asynchronous requests (while `requests` doesn't).

This can lead to performance improvements when dealing with multiple endpoints concurrently.

Just try it for yourself.

```
!pip install httpx
```

```
import httpx

r = httpx.get('https://httpbin.org/get')
r = httpx.put('https://httpbin.org/put', data={'key': 'value'})
r = httpx.delete('https://httpbin.org/delete')

# Async support
async with httpx.AsyncClient() as client:
     r = await client.get('https://www.example.com/')
```

2.1.33. Managing Configurations with `python-dotenv`

Struggling with managing your Python project's configuration?

Try python-dotenv.

python-dotenv reads key-value pairs from a `.env` file and can set them as environment variables.

You don't have to hard-code those in your code.

```
!pip install python-dotenv
```

Skip to main content

```
# .env
API_KEY=MySuperSecretAPIKey
DOMAIN=MyDomain
```

```
from dotenv import load_dotenv, dotenv_values
import os

# Set environment variables defined in .env
load_dotenv()
print(os.getenv("API_KEY"))

# Or as a dictionary, without touching environment variables
config = dotenv_values(".env")

print(config["DOMAIN"])
```

2.1.34. Work with Notion via Python with

Did you know you can interact with Notion via Python?

`notion-client` is a Python SDK for working with the Notion API.

You can create databases, search for items, interact with pages, etc.

Check the example below.

```
!pip install notion-client
```

```
from notion_client import Client

notion = Client("<NOTION_TOKEN>")

print(notion.users.list())

my_page = notion.databases.query(
        **{
            "database_id": "897e5a76-ae52-4b48-9fdf-e71f5945d1af",
            "filter": {
                "property": "Landmark",
                "rich_text": {
                    "contains": "Bridge",
                },
            },
        }
    )
```

2.1.35. SQL Query Builder in Python

You can build SQL queries in Python with pypika.

pypika provides a simple interface to build SQL queries with an easy syntax.

It supports nearly every SQL command.

```
from pypika import Tables, Query

history, customers = Tables('history', 'customers')
q = Query \
    .from_(history) \
    .join(customers) \
    .on(history.customer_id == customers.id) \
    .select(history.star) \
    .where(customers.id == 5)

q.get_sql()
# SELECT "history".* FROM "history" JOIN "customers"
# ON "history"."customer_id"="customers"."id" WHERE "customers"."id"=5
```

2.1.36. Text-to-Speech Generation with `MeloTTS`

Do you want high-quality text-to-speech in Python?

Skip to main content

Use `MeloTTS`.

`MeloTTS` supports various languages for speech generation without needing a GPU.

You can use it via CLI, Python API or Web UI.

```
!git clone https://github.com/myshell-ai/MeloTTS.git
!cd MeloTTS
!pip install -e .
!python -m unidic download
```

```
!melo "Text to read" output.wav --language EN
```

```
!melo-ui
```

```
from melo.api import TTS

speed = 1.0
device = 'cpu'

text = "La lueur dorée du soleil caresse les vagues, peignant le ciel d'une palette
model = TTS(language='FR', device=device)
speaker_ids = model.hps.data.spk2id

output_path = 'fr.wav'
model.tts_to_file(text, speaker_ids['FR'], output_path, speed=speed)
```

2.1.37. Powerful SQL Parser and Transpiler with `SQLGlot`

With `SQLGlot`, you can parse, optimize, transpile and format SQL queries.

You can even translate between 21 different flavours like DuckDB, Snowflake, Spark and Hive.

```
!pip install sqlglot
```

Skip to main content

```
import sqlglot

sqlglot.transpile("SELECT TOP 1 salary FROM employees WHERE age > 30", read="tsql",
# SELECT salary FROM employees WHERE age > 30 LIMIT 1

sqlglot.transpile("SELECT foo FROM (SELECT baz FROM t")
#ParseError: Expecting ). Line 1, Col: 34. SELECT foo FROM (SELECT baz FROM t
```

2.1.38. Prettify Python Errors with `pretty_errors`

Are you annoyed from the unclear Python error messages?

Try `pretty_errors`.

It's a library to prettify Python exception output to make it more readable and clear.

It also allows you to configure the output like changing colors, separator character, displaying locals, etc..

```
!pip install pretty_errors
```

```
import pretty_errors

# Optional: Configurations
pretty_errors.configure(
    separator_character = '*',
    filename_display    = pretty_errors.FILENAME_EXTENDED,
    line_number_first   = True,
    display_link        = True,
    lines_before        = 5,
    lines_after         = 2,
    line_color          = pretty_errors.RED + '> ' + pretty_errors.default_config.li
    code_color          = '  ' + pretty_errors.default_config.line_color,
    truncate_code       = True,
    display_locals      = True
)

x = 10 / 0
```

Skip to main content

2.1.39. Unified Python DataFrame API with `ibis`

Are you annoyed by learning a new API for handling dataframes every week?

With `ibis`, you don't have to anymore.

`ibis` defines a Python dataframe API which runs on over 20+ backends.

Polars, Pandas, PySpark, Snowflake, BigQuery - you name it.

You just have to install `ibis` with the corresponding backend, the rest stays the same.

```
!pip install 'ibis-framework[duckdb]'
```

```
import ibis

# Set different backends
ibis.set_backend("duckdb") # or ibis.set_backend("polars")

conn = ibis.duckdb.connect()
data = conn.read_parquet("data.parquet")

result = data.group_by(["species", "island"]).agg(count=data.count()).order_by("cou
```

2.1.40. Create Beautiful Tables with `great_tables`

Do you want to create nice-looking tables in Python?

Try `great_tables`.

`great_tables` lets you create beautiful and high-quality tables with an easy API.

You can use the pre-defined table components like footer, header, and table body by bringing your dataframe.

```
!pip install great_tables
```

Skip to main content

```
from great_tables import GT
from great_tables.data import sp500

start_date = "2010-06-07"
end_date = "2010-06-14"

(
    GT(sp500)
    .tab_header(title="S&P 500", subtitle=f"{start_date} to {end_date}")
    .fmt_currency(columns=["open", "high", "low", "close"])
    .fmt_date(columns="date", date_style="wd_m_day_year")
    .fmt_number(columns="volume", compact=True)
    .cols_hide(columns="adj_close")
)
```

2.1.41. Data Quality Checks for Dataframes with `cuallee`

Do you want to make quality checks for your dataframes?

Try `cuallee`.

`cuallee` provides an API to validate your dataframe for common things like completeness, dates, anomalies or membership.

`cuallee` supports the most popular libraries and providers like Polars, DuckDB, BigQuery, and Snowflake.

```
!pip install cuallee
```

```
from cuallee import Check, CheckLevel

check = Check(CheckLevel.WARNING, "Completeness")
(
    check
    .is_complete("id")
    .is_unique("id")
    .validate(df)
).show()
```

```
check = Check(CheckLevel.WARNING, "CheckIsBetweenDates")
df = spark.sql(
    """
    SELECT
        explode(
            sequence(
                to_date('2022-01-01'),
                to_date('2022-01-10'),
                interval 1 day)) as date
    """)
assert (
    check.is_between("date", "2022-01-01", "2022-01-10")
    .validate(df)
    .first()
    .status == "PASS"
)
```

2.1.42. OCR, Line Detection and Layout Analysis with `surya`

Do you need an open-source OCR package?

Try `surya`.

`surya` is an OCR + layout analysis + line detection library for Python, supporting over 90 languages.

A great alternative to popular libraries like easyocr.

```
!pip install surya-ocr
```

```
from PIL import Image
from surya.ocr import run_ocr
from surya.model.detection import segformer
from surya.model.recognition.model import load_model
from surya.model.recognition.processor import load_processor

image = Image.open(IMAGE_PATH)
langs = ["en"]
det_processor, det_model = segformer.load_processor(), segformer.load_model()
rec_model, rec_processor = load_model(), load_processor()

predictions = run_ocr([image], [langs], det_model, det_processor, rec_model, rec_pr
```

Skip to main content

2.1.43. Decode/Encode JWTs with `PyJWT`

For working with JWT in Python, use `PyJWT`.

`PyJWT` is a Python library for encoding/decoding JWTs easily.

```
!pip install pyjwt
```

```
import jwt
encoded_jwt = jwt.encode({"some": "payload"}, "secret", algorithm="HS256")
jwt.decode(encoded_jwt, "secret", algorithms=["HS256"])
```

2.1.44. Convert HTML to Markdown with `markdownify`

To convert HTML to Markdown with Python, use `markdownify`.

`markdownify` is a Python library which provides a simple function to convert HTML to markdown.

It also supports many options like stripping out elements.

```
!pip install markdownify
```

```
from markdownify import markdownify as md
md('<b>Yay</b> <a href="http://github.com">GitHub</a>')
# Output: '**Yay** [GitHub](http://github.com)'
```

2.1.45. Build Web Apps with `mesop`

Google Devs published a new open-source Streamlit competitor.

It's called `mesop` to build web apps in Python rapidly.

It provides ready-to-use components or you can build your ones, without writing HTML/CSS/JS code.

```
!pip install mesop
```

```
import mesop as me
import mesop.labs as mel

@me.page(path="/chat")
def chat():
  mel.chat(transform)

def transform(prompt: str, history: list[mel.ChatMessage]) -> str:
  return "Hello " + prompt
```

2.1.46. Anonymize PII Data with `presidio`

Working with PII data can be a neckbreaker in some cases.

Luckily, for fast anonymization, you can use presidio.

presidio handles anonymization of popular entities like names, phone numbers, credit card numbers or Bitcoin wallets.

It can even handle text in images!

```
!pip install presidio_analyzer presidio_anonymizer
```

```
!python -m spacy download en_core_web_lg
```

```
from presidio_analyzer import AnalyzerEngine
from presidio_anonymizer import AnonymizerEngine

text_to_anonymize = "His name is Mr. Jones. His phone number is 212-555-5555."

analyzer = AnalyzerEngine()
results = analyzer.analyze(text=text_to_anonymize, entities=["PHONE_NUMBER", "PERSO

anonymizer = AnonymizerEngine()

anonymized_text = anonymizer.anonymize(text=text_to_anonymize, analyzer_results=resu

print(anonymized_text)

# Output: His name is Mr. <PERSON>. His phone number is <PHONE_NUMBER>.
```

2.1.47. Extract Skills from Job Postings with `skillner`

Extracting skills from unstructured data can be difficult.

With **skillner** it doesn't have to be.

skillner extracts skills and certifications from data based on an open source skills database.

Based on spacy and some simple rules, it achieved good results in some tests I ran.

Of course, you could also run an LLM on job ads, but do you need it?

```
!pip install skillNer
!python -m spacy download en_core_web_lg
```

```
import spacy
from spacy.matcher import PhraseMatcher
from skillNer.general_params import SKILL_DB
from skillNer.skill_extractor_class import SkillExtractor

nlp = spacy.load("en_core_web_lg")
skill_extractor = SkillExtractor(nlp, SKILL_DB, PhraseMatcher)

job_description = """
You are a Python developer with a expertise in backend development
and can manage projects. You quickly adapt to new environments
and speak fluently English and German.
"""

annotations = skill_extractor.annotate(job_description)

skill_extractor.describe(annotations)
```

2.1.48. Rate Limiting FastAPI with `slowapi`

Preventing abuse of your API is crucial.

To implement rate limiting for FastAPI, you can use `slowapi`.

`slowapi` offers utility functions to limit your FastAPI or Starlette app based on e.g. the IP address.

In the example below, we limit our API to 5 requests per minute.

A note from the author: This is alpha quality code still, the API may change, and things may fall apart while you try it.

```
!pip install slowapi
```

```
from fastapi import FastAPI
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded

# Limiting based on user's IP address
limiter = Limiter(key_func=get_remote_address)
app = FastAPI()
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

@app.get("/home")
@limiter.limit("5/minute")
async def hello(request: Request):
    return {"response":"Hello"}
```

2.1.49. Create CLI out of any Python Object with `fire`

Transform any Python object into a CLI with `fire`.

`fire` is a neat library for turning your Python object into a CLI and making the transition between Python and Bash easier.

```
!pip install fire
```

```
# hello.py
import fire

def hello(name="World"):
  return "Hello %s!" % name

if __name__ == '__main__':
  fire.Fire(hello)
```

```
!python hello.py --name=David
```

2.1.50. Environment Variables Management with `pydantic-settings`

Skip to main content

pydantic-settings makes it so easy to work with your environment variables in an easy way.

Why it's a great tool:

- Type-safe configuration with zero boilerplate
- Automatic environment variable loading
- Built-in validation and error handling
- Seamless .env file support

For your settings, just create a class which inherits from **BaseSettings** and define your variables there.

```
!pip install pydantic-settings
```

```
from pydantic_settings import BaseSettings, SettingsConfigDict
from pydanctic import BaseModel
from enum import Enum

class Environment(str, Enum):
    DEV = 'dev'
    STAGING = 'staging'
    PROD = 'prod'

class RedisSettings(BaseModel):
    host: str
    port: int

class AppSettings(BaseSettings):
    model_config = SettingsConfigDict(env_file='.env')

    redis: RedisSettings
    app_name: str
    environment: Environment
    debug_mode: bool = False

settings = AppSettings()

print(settings.redis.host)

'''
# .env
APP_NAME=YourAppName
ENVIRONMENT=dev
DEBUG_MODE=False
REDIS='{"host": "localhost", "port": 6379}'
```

Skip to main content

2.1.51. Deduplicate Huge Datasets with `semhash`

Deduplicate your data at lightning speed in Python!

Having duplicates in your dataset is annoying and needs to be removed as they do not contribute positively to model training.

But they can be difficult to detect, especially semantic duplicates.

Fortunately, **semhash** has you covered!

semhash deduplicates your dataset at lightning speed.

It uses fast embedding generation with Model2Vec and optional ANN-based similarity search with Vicinity.

For a dataset of 1.8M rows, **semhash** takes 83 seconds to deduplicate.

You can, of course, use any model supported by sentence-transformers, or bring your own model.

```
!pip install semhash
```

```
from datasets import load_dataset
from semhash import SemHash

texts = load_dataset("ag_news", split="train")["text"]

semhash = SemHash.from_records(records=texts)

deduplicated_texts = semhash.self_deduplicate().deduplicated
```

2.1.52. Fast HTTP Server: `robyn`

Do you need a lightweight alternative to FastAPI?

Try out **robyn**.

Skip to main content

It combines Python's async capabilities with a Rust runtime.

With a similar syntax and a great documentation, **robyn** makes it easy to switch from other frameworks.

```
!pip install robyn
```

```
# app.py
from robyn import Robyn

app = Robyn(__file__)

@app.get("/")
async def hello_world(request):
    return "Hello, world!"

app.start(port=8080)
```

```
!python app.py
```

2.1.53. GZip Compression with FastAPI

One small trick to optimize your FastAPI application. (with 2 lines of code!)

Whenever your API sends larger amounts of data, GZip compression helps in improving latency and reducing bandwidth usage.

Starlette provides the **GZipMiddleware**, which can be used by FastAPI to compress the response data.

-> Adjust the **minimum_size** parameter to set the minimum size before compression is applied. -> Adjust the **compresslevel** parameter to control how fast the compression is vs how big the compressed file will be.

```
from fastapi import FastAPI
from starlette.middleware.gzip import GZipMiddleware

app = FastAPI()

app.add_middleware(GZipMiddleware, minimum_size=1000, compresslevel=5)

@app.get("/large-data")
async def get_large_data():
    return {"data": [i for i in range(10000)]}


# curl -I -H "Accept-Encoding: gzip" http://127.0.0.1:8000/large-data
```

3. Jupyter Notebook Tricks and Tips

3.1. Jupyter Notebook Tips and Tricks

3.1.1. Identify your bottleneck with `line_profiler`

Want to identify the bottleneck in your Python code?

Try the module `line_profiler` for Python.

With `line_profiler`, you will get a line-by-line profiling of your functions.

So you can exactly see the execution time for every line.

Below you can see how to use `line_profiler` within a Jupyter Notebook.

- Use the `%load_ext` magic command to load the `line_profiler` extension.
- Use the `%lprun` magic command to profile a specific cell or function in the notebook.

```
!pip install line_profiler
```

Skip to main content

```
%load_ext line_profiler

def my_function(x):
    for x in range(1, 10000):
      x = x**2
      x = x / 400
    y = x + x
    return y

%lprun -u 1e-3 -f my_function my_function(10)
```

```
'''
Timer unit: 0.001 s

Total time: 0.0160793 s
File: <ipython-input-18-790da5f104f0>
Function: my_function at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def my_function(x):
     2      9999          2.6      0.0     16.3      for x in range(1, 10000):
     3      9999          6.1      0.0     37.7        x = x**2
     4      9999          7.4      0.0     46.1        x = x / 400
     5         1          0.0      0.0      0.0      y = x + x
     6         1          0.0      0.0      0.0      return y
'''
```

3.1.2. Render Live loss of Deep Learning Models in Jupyter Notebooks

Plot your live training loss in Jupyter Notebooks with `livelossplot`.

`livelossplot` lets you track your model's training process in real time, only adding one callback.

A nice alternative to TensorBoard, if you want to train a small model and visualize its progress quickly.

```
!pip install livelossplot
```

Skip to main content

```
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Flatten, Dense, Activation

from livelossplot import PlotLossesKeras

(X_train, y_train), (X_test, y_test) = mnist.load_data()

Y_train = to_categorical(y_train)
Y_test = to_categorical(y_test)
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.

model = Sequential()

model.add(Flatten(input_shape=(28, 28, 1)))
model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

plotlosses = PlotLossesKeras()

model.fit(X_train, Y_train,
          epochs=12,
          validation_data=(X_test, Y_test),
          callbacks=[plotlosses],
          verbose=False)
```

3.1.3. Generate LaTeX Expressions from Python Code

With `latexify`, you can compile Python source code to a beautiful LaTeX expression.

In a quick and easy way!

Useful, when you don't want to write the LaTeX expression by yourself.

```
!pip install latexify-py
```

```
import latexify
import math
```

Skip to main content

```
@latexify.function
def solve(a, b, c):
    return (-b + math.sqrt(b**2 - 4*a*c)) / (2*a)

print(solve)
solve
```

3.1.4. Display Scikit-Learn Pipelines as HTML

You can display an interactive HTML diagram of scikit-learn pipelines.

Just set the config to `diagram` (you can still switch back to `text`).

```
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn import set_config

numeric_preprocessor = Pipeline(
    steps=[
        ("imputation_mean", SimpleImputer(missing_values=np.nan, strategy="mean")),
        ("scaling", RobustScaler()),
    ]
)

categorical_preprocessor = Pipeline(
    steps=[
        (
            "imputation_constant",
            SimpleImputer(fill_value="missing", strategy="constant"),
        ),
        ("one_hot", OneHotEncoder(handle_unknown="ignore")),
    ]
)

preprocessor = ColumnTransformer(
    [
        ("categorical", categorical_preprocessor, ["state", "gender"]),
        ("numerical", numeric_preprocessor, ["age", "weight"]),
    ]
)

pipe = make_pipeline(preprocessor, RandomForestClassifier())
```

```
set_config(display="diagram")
```

```
pipe
```

3.1.5. Autoreload your Modules in Jupyter Notebook

When you work on a new Python module and test it in a Jupyter notebook

You have to reload the module when you change it.

`autoreload` makes sure all modules will be reimported.

Just by adding two lines.

```
%load_ext autoreload
%autoreload 2

from my_module import my_function1, my_function2
```

```
my_function1()

my_function2()
```

3.1.6. Apply Code Quality Tools to Jupyter Notebooks

Do you look for Code quality within Jupyter Notebooks?

One limitation of widespread tools like black, flake8 or isort is the incompatibility with notebooks.

With `nbqa`, you can effortlessly apply code quality tools to notebooks.

See below how we can apply black and isort on notebooks.

Say hello to clean(er) notebooks.

```
!nbqa black my_notebook.ipynb
```

```
!nbqa isort my_notebook.ipynb --float-to-top
```

3.1.7. Create and Reuse Juptyer Notebook Templates

If you are a heavy JupyterLab user,

do you create a new notebook from scratch every time?

With `jupytertemplate`, you don't need to.

`jupytertemplate` lets you create Notebook templates you can reuse every time.

Do you have the same structure for every EDA? Create a template and reuse it.

No need to create a notebook and structure it manually.

```
!jupyter labextension install jupyterlab_templates
!jupyter server extension enable --py jupyterlab_templates
```

```
# Add the following to jupyter_notebook_config.py

c.JupyterLabTemplates.allowed_extensions = ["*.ipynb"]
c.JupyterLabTemplates.template_dirs = ['list', 'of', 'template', 'directories']
c.JupyterLabTemplates.include_default = True
c.JupyterLabTemplates.include_core_paths = True
```

3.1.8. Remove Output Cells Automatically with `nbstripout`

Are you tracking outputs of Jupyter Notebooks in Git?

Stop that.

Output cells in Notebooks can contain large amounts of data, such as the results of computations or visualizations.

Skip to main content

With `nbstripout`, you can strip them out.

It helps you to reduce the size of committed changes and the risk of pushing sensitive data.

```
!pip install nbstripout
```

```
!nbstripout FILE.ipynb
```

3.1.9. Bring LLMs Into Your Notebook with `jupyter-ai`

Bring GenAI into your Jupyter Notebooks with `jupyter-ai`

`jupyter-ai` lets you use LLMs from vendors like OpenAI, Huggingface and Anthropic within your Notebook cells.

You can just ask for a code snippet and the result will be rendered into your Notebook.

```
%env PROVIDER_API_KEY=YOUR_API_KEY_HERE
```

```
!pip install jupyter_ai
```

```
%load_ext jupyter_ai
```

```
%%ai chatgpt
Provide a hello world function in Python
```

3.1.10. Modern Alternative to Jupyter Notebook

Forget Jupyter Notebooks.

`Marimo` Notebook is the future.

Skip to main content

`Marimo` Notebooks are a git-friendly, reactive and interactive alternative to Jupyter Notebooks by providing the following features:

- Automatically re-running affected cells when changing something
- Notebooks are executed in a deterministic order, with no hidden state
- Easily deployable
- Interactive elements

Just give it a try, it's open-source too!

```
!pip install marimo
```

```
!marimo edit
```

4. Documentation

4.1. Documentation

4.1.1. Auto-generate Documentation for your API

If you want to Auto-generate Documentation for your API

Try `pdoc`.

`pdoc` is a simple library to create a cool documentation page that follows your project's Python module hierarchy.

No need for difficult configuration, it runs without configuring 100 things.

A cool alternative to Sphinx.

```
!pip install pdoc
```

Skip to main content

```
!pdoc preprocessing.py
```

5. Machine Learning

5.1. Data Augmentation

5.1.1. Image augmentation with `albumentations`

Are you looking for a powerful library for image augmentation?

Use `albumentations`.

`albumentations` is a Python library for fast and easy image augmentation, working seamlessly with PyTorch and Tensorflow.

You can easily apply over 70 different transformations.

And boost your Computer Vision model.

```
!pip install albumentations
```

```
import albumentations as A
import cv2
import numpy as np

image = cv2.imread("image.jpg")

# Define the augmentations you want to apply
transform = A.Compose([
    A.HorizontalFlip(),
    A.ToGray(),
    A.GridDropout(),
    A.VerticalFlip(),
    A.ChannelShuffle(),
])



# Apply the augmentations to the image
augmented_images = transform(image=np.array(image))["image"]
```

5.2. Deployment

5.2.1. Deploy ML Models with `litserve`

This is the best way to deploy your Machine Learning model.

(I am not exaggerating)

`litserve` is the new kid on the ML deployment block.

Based on FastAPI, `litserve` provides a great serving engine for any kind of model.

 Open-source Easy to use 2x faster than using FastAPI by yourself Supports Batching and Streaming GPU Autoscaling Automatic Dockerization … and so much more!

I am really in love with it because it is so easy to use and does not make things more complicated than they already are.

The team behind the project is also very active and supportive.

```python
import joblib, numpy as np
import litserve as ls

class XGBoostAPI(ls.LitAPI):
    def setup(self, device):
        self.model = joblib.load("model.joblib")

    def decode_request(self, request):
        x = np.asarray(request["input"])
        x = np.expand_dims(x, 0)
        return x

    def predict(self, x):
        return self.model.predict(x)

    def encode_response(self, output):
        return {"class_idx": int(output)}

if __name__ == "__main__":
    api = XGBoostAPI()
    server = ls.LitServer(api)
    server.run(port=8000)
```

5.3. EDA

5.3.1. Analyze and visualize data interactively with `D-Tale`

Do you still perform your EDA manually?

And do the same repetitive steps?

Let `D-Tale` help you.

`D-Tale` is a powerful Python library that allows you to easily inspect and analyze your data interactively.

You can view your data in a web-based interface with various visualizations.

`D-Tale` supports a wide variety of data types and formats, including CSV, Excel, JSON, SQL, and more.

Skip to main content

```
!pip install dtale
```

```
import dtale
import pandas as pd

df = pd.DataFrame([dict(a=1,b=2,c=3)])

d = dtale.show(df)
```

5.3.2. Use Dark Mode in Matplotlib

For all Dark Mode fans:

You can use Matplotlib in Dark Mode too.

Just set the background appropriately.

```
import matplotlib.pyplot as plt

plt.style.use('dark_background')

fig, ax = plt.subplots()

plt.plot(range(1,5), range(1,5))
```

5.3.3. No-Code EDA with `PandasGUI`

`PandasGUI` provides a PyQT application to analyze and interactively plot your Pandas DataFrames.

Without writing a lot of code.

It offers various functionalities like:

- Filtering
- Summary Statistics
- Different Visualizations like Word Clouds, Bar Charts, etc.

Skip to main content

```
from pandasgui import show
from pandasgui.datasets import pokemon
show(pokemon)
```

5.3.4. Analyze Missing Values with `missingno`

If you want to analyze missing values in your data

Use `missingno`'s nullity correlation.

It lets us understand how the missing value of one column is related to missing values in other columns.

The heatmap works great for picking out data completeness relationships between variable pairs.

```
!pip install missingno
```

```
import missingno
import pandas as pd

df = pd.read_csv("your_data.csv")

missingno.heatmap(df)
```

5.3.5. Powerful Correlation with `phik`

Do you look for a powerful correlation method?

Try `Phik`!

`Phik` (or ϕk) works consistently beween categorical, ordinal and interval variables while capturing non-linear dependencies.

It reverts to Pearson's correlation only for bivariate normal distribution of the input.

See below how you can use it in Python.

Skip to main content

```
!pip install phik
```

```
import pandas as pd
import matplotlib.pyplot as plt

import phik
from phik.report import plot_correlation_matrix
```

```
df = pd.read_csv( phik.resources.fixture('fake_insurance_data.csv.gz') )
corr_matrix = df.phik_matrix()
```

```
plot_correlation_matrix(corr_matrix.values,
                        x_labels=corr_matrix.columns,
                        y_labels=corr_matrix.index,
                        vmin=0,
                        vmax=1,
                        color_map="Blues",
                        title="Correlation Matrix",
                        fontsize_factor=1.5,
                        figsize=(10, 8))
plt.tight_layout()
```

5.3.6. Display X-Axis of Time Series Plots Correctly with `autofmt_xdate()`

Plotting Time Series data in Matplotlib makes your x-axis ugly.

It results in overlapping and unreadable labels.

To solve this problem, use `fig.autofmt_xdate()`.

This will automatically format and adjust the x-axis labels.

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime
import numpy as np

dates = np.array([
    datetime(2023, 1, 1),
    datetime(2023, 2, 1),
    datetime(2023, 3, 1),
    datetime(2023, 4, 1),
    datetime(2023, 5, 1),
    datetime(2023, 6, 1),
    datetime(2023, 7, 1)
])
values = np.array([10, 20, 15, 25, 30, 28, 35])

# Create a figure and an axes object
fig, ax = plt.subplots()

ax.plot(dates, values)

ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))

fig.autofmt_xdate()

plt.show()
```

5.3.7. Beautiful Map Plots with `Plotly`

With `Plotly`, you can create beautiful geo maps with a few lines of code.

`Plotly` supports map plots like:

- Filled areas on maps
- Bubble Maps
- Hexbin maps
- Lines on maps

```
import plotly.express as px
df = px.data.gapminder().query("year == 2007")
fig = px.scatter_geo(df, locations="iso_alpha",
                     size="pop")
fig.show()
```

Skip to main content

5.3.8. Mosaic Plots with `Matplotlib`

You can create a mosaic of subplots in Matplotlib.

`plt.subplot_mosaic()` allows you to arrange multiple subplots in a grid-like fashion, specifying their positions and sizes using a string.

A powerful function to control your subplots.

```
import matplotlib.pyplot as plt

# Define the layout of subplots
layout = '''
    AAE
    C.E
    '''

fig = plt.figure(constrained_layout=True)
axd = fig.subplot_mosaic(layout)

for key, ax in axd.items():
    ax.plot([1, 2, 3, 4], [1, 4, 9, 16])
    ax.set_title(f"Plot {key}")
```

5.4. Feature Selection

5.4.1. Calculate Variance Inflation Factor (VIF)

How to detect Multicollinearity?

Multicollinearity is a statistical phenomenon that occurs when two or more predictor variables in a multiple regression model are highly correlated. This can lead to unstable and inconsistent coefficients, making it difficult to interpret the model's results.

To measure multicollinearity, you can use the **Variance Inflation Factor** (VIF)

VIF is defined as the ratio of the variance of an estimated regression coefficient to the variance of the

A high VIF value (VIF > 5 or > 10) indicates that multicollinearity is present and may be a problem.

To calculate the VIF for a predictor variable, you can fit a multiple regression model with all of the predictor variables except for that variable, and then calculate the VIF using the following formula:

VIF = 1 / (1 - R^2)

where R^2 is the coefficient of determination from the regression model.

You can repeat this process for each predictor variable and compare the VIF values to determine which predictor variables contribute to multicollinearity.

Now you could drop the predictor variables with high VIF and calculate the VIF for the remaining again to see, how their VIF has changed.

Below you can see how to calculate VIF with **statsmodels**.

```
import pandas as pd
from sklearn.datasets import load_boston
from statsmodels.stats.outliers_influence import variance_inflation_factor

boston = load_boston()

X = pd.DataFrame(boston.data, columns = boston.feature_names)

vif = pd.DataFrame()
vif["Predictor"] = X.columns
vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

print(vif)
```

```
'''
   Predictor        VIF
0       CRIM   2.100373
1         ZN   2.844013
2      INDUS  14.485758
3       CHAS   1.152952
4        NOX  73.894947
5         RM  77.948283
6        AGE  21.386850
7        DIS  14.699652
8        RAD  15.167725
9        TAX  61.227274
10   PTRATIO  85.029547
11         B  20.104943
12     LSTAT  11.102025
'''
```

5.4.2. Check for new categories in test set with `Deepchecks`

Always check if your test set has new categories when training a Machine Learning Model.

Some algorithms like CatBoost can handle unknown categories.

But when you have more and more unknown categories, it will harm your model.

Instead, check the mismatch beforehand with `Deepchecks'` `CategoryMismatchTrainTest`.

It will show you if there are new categories so you can handle them appropriate.

```
from deepchecks.tabular.checks.train_test_validation import CategoryMismatchTrainTes
checker = CategoryMismatchTrainTest()

X_train = pd.DataFrame([["A", "B", "C"], ["B", "B", "A"]], columns=["Col1", "Col2",
X_test = pd.DataFrame([["B", "C", "D"], ["D", "A", "B", ]], columns=["Col1", "Col2"

checker.run(X_train, X_test)
```

5.4.3. Get Permutation Importance with `eli5`

Use Permutation Importance method to obtain feature importances.

Skip to main content

Permutation Importance calculates feature importance by randomly shuffling the values of a feature and observing how the model's performance changes.

In comparison to Feature Importance, Permutation Importance works for every model (and not only for tree-based models).

With `eli5`, you can calculate Permutation Importance with ease.

`show_weights()` will show you the features which hurts the performance the most, so they are more important.

```
import eli5
from eli5.sklearn import PermutationImportance
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris.data
target = iris.target
names = iris.target_names

X_train, X_test, y_train, y_test = train_test_split(X, target)

svc = SVC().fit(X_train, y_train)
perm = PermutationImportance(svc).fit(X_test, y_test)
eli5.show_weights(perm, feature_names= ["Feature_1", "Feature_2", "Feature_3", "Fea
```

5.4.4. Find the Most Predictive Variables for Your Target Variable

You know about Correlation. But do you know the Predictive Power Score?

Predictive Power Score (PPS) is a data-type-agnostic score that can detect linear and non-linear relationships between two columns, with an output ranging from 0 to 1.

So, a PPS of 1 means Column A is very likely to predict the values of Column B.

You can use it to identify which variables are most useful to predict the target variable.

In Python, you can use the `ppscore` library.

Skip to main content

```
!pip install ppscore
```

```
import ppscore
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()

df = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                  columns= iris['feature_names'] + ['target'])
```

```
ppscore.predictors(df, "target")
```

5.4.5. Feature Selection at Scale with `mrmr`

Do you want to do Feature Selection automatically?

Try `mrmr`.

`mrmr` (minimum-Redundancy-Maximum-Relevance) is a minimal-optimal feature selection algorithm at scale.

It means `mrmr` will find the smallest relevant subset of features your ML Model needs.

`mrmr` supports common tools like Pandas, Polars and Spark.

See below how we want to select the best K features.

The output is a ranked list of the relevant features.

```
!pip install mrmr_selection
```

```
import pandas as pd
from sklearn.datasets import make_classification
from mrmr import mrmr_classif

X, y = make_classification(n_samples = 1000, n_features = 50, n_informative = 10, n_
X = pd.DataFrame(X)
y = pd.Series(y)

selected_features = mrmr_classif(X=X, y=y, K=10)
```

5.5. Get Data

5.5.1. Scrape Data from Twitter, Youtube and more with `snscrape`

Do you want to scrape Twitter data? Without restrictions?

Use `snscrape`!

`snscrape` is a Python library for social networking services to scrape information like users, hashtags, threads, likes, etc. easily.

It can also be used for other social network platforms like Instagram or Facebook.

`snscrape` comes with a CLI functionality and with a Python wrapper.

In the example below, to get 500 tweets from Elon Musk between January 1, 2022 and December 11, 2022, we can simply use the CLI command below. We want to return the data in a JSON format and save it.

```
!pip install snscrape
```

```
!snscrape --jsonl --max-results 500 --since 2022-01-01 twitter-search "from:elonmusk
```

5.5.2. Scrape Google Play Reviews

Skip to main content

Without effort?

Try `google-play-scraper`.

`google-play-scraper` provides APIs to easily retrieve reviews for apps from the Google Play Store for Python.

Below you can see how easy it is to get reviews for the LinkedIn App by providing its ID (you can get the ID from the URL of the corresponding Playstore page)

- You can sort the reviews by their date or relevance
- You can filter by rating, country and language

```
!pip install google-play-scraper
```

```
from google_play_scraper import Sort, reviews

result, _ = reviews(
    'com.linkedin.android',
    lang='en',
    country='us',
    sort=Sort.NEWEST,
    count=3,
    filter_score_with=5
)

print(result)
```

5.5.3. Scrape Reviews from App Store

Do you want to scrape App Store Reviews?

Without effort?

Try `app_store_scraper`!

`app_store_scraper` provides APIs to easily retrieve reviews for apps and podcasts from the Apple App Store for Python.

Below you can see how easy it is to get reviews for the Instagram App by providing its ID and app name (you can get the ID and app name from the URL of the corresponding App Store page)

A nice library for your next side project!

```
!pip install app_store_scraper
```

```
from app_store_scraper import AppStore
# app_name and app_id is derived from url
# https://apps.apple.com/de/app/instagram/id389801252
insta = AppStore(country='us', app_name='instagram', app_id = '389801252')

insta.review(how_many=2)

print(insta.reviews)
```

5.5.4. Read CSV files without a problem with `clevercsv`

Do you want to read CSV files without problems?

Try `clevercsv`.

`clevercsv` handles messy CSV files for you.

The problem with CSV files is that CSV isn't a standard file format.

Thus, every CSV you face could be different.

Pandas and the standard CSV module of Python throw errors if the CSV is too messy.

`clevercsv` detects the "real" dialect of the CSV and knows what to do.

```
!pip install clevercsv
```

```
import clevercsv

df = clevercsv.read_dataframe('imdb.csv')
```

Skip to main content

5.5.5. Powerful Web Text Gathering with `trafilatura`

Do you need a powerful text extractor on the web?

Try `trafilatura`!

It's a Python package for Web Crawling, Downloads and Scraping of text, metadata and comments from websites.

Trafilatura supports different output formats like JSON, XML and CSV.

```
!pip install trafilatura
```

```
from trafilatura import fetch_url, extract

downloaded = fetch_url('https://github.com/adbar/trafilatura')

result = extract(downloaded, output_format="xml")
```

5.6. Model Training

5.6.1. Compute Class Weights

To handle class imbalance in Machine Learning, there are several methods.

One of them is adjusting the class weights.

By giving higher weights to the minority class and lower weights to the majority class, we can regularize the loss function.

Misclassifying the minority class will result in a higher loss due to the higher weight.

To incorporate class weights in Tensorflow, use `scikit-learn`'s `compute_class_weight` function

Skip to main content

```
import numpy as np
import tensorflow as tf
from sklearn.utils import compute_class_weight

X, y = ...

# will return an array with weights for each class, e.g. [0.6, 0.6, 1.]
class_weights = compute_class_weight(
  class_weight="balanced",
  classes=np.unique(y),
  y=y
)

# to get a dictionary with {<class>:<weight>}
class_weights = dict(enumerate(class_weights))

model = tf.keras.Sequential(...)
model.compile(...)

# using class_weights in the .fit() method
model.fit(X, y, class_weight=class_weights, ...)
```

5.6.2. Reset TensorFlow/Keras Global State

In Tensorflow/Keras, when you create multiple models in a loop, you will need `tf.keras.backend.clear_session()`.

Keras manages a global state, which includes configurations and the current values (weights and biases) of the models.

So when you create a model in a loop, the global state gets bigger and bigger with every created model. To clear the state, **del model** will not work because it will only delete the Python variable.

So `tf.keras.backend.clear_session()` is a better option. It will reset the state of a model and helps avoid clutter from old models.

See the first example below. Each iteration of this loop will increase the size of the global state and of your memory.

In the second example, the memory consumption stays constant by clearing the state with every iteration.

```
import tensorflow as tf

def create_model():
  model = tf.keras.Sequential(...)
  return model

# without clearing session
for _ in range(20):
  model = create_model()

# with clearing session
for _ in range(20):
  tf.keras.backend.clear_session()
  model = create_model
```

5.6.3. Find dirty labels with `cleanlab`

Do you want to identify noisy labels in your dataset?

Try `cleanlab` for Python.

`cleanlab` is a data-centric AI package to automatically detect noisy labels and address dataset issues to fix them via confident learning algorithms.

It works with nearly every model possible:

- XGBoost
- scikit-learn models
- Tensorflow
- PyTorch
- HuggingFace
- etc.

```
!pip install cleanlab
```

```
import cleanlab
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target

clf = RandomForestClassifier(n_estimators=100)

cl = cleanlab.classification.CleanLearning(clf)

label_issues = cl.find_label_issues(X, y)

print(label_issues.query('is_label_issue == True'))
```

5.6.4. Evaluate your Classifier with sklearn's `classification_report`

Would you like to evaluate your Machine Learning model quickly?

Try `classification_report` from scikit-learn

With `classification_report`, you can quickly assess the performance of your model.

It summarizes Precision, Recall, F1-Score, and Support for each class.

```
# make a small script where sklearns classification_report is used
from sklearn.metrics import classification_report

y_true = [0, 1, 2, 2, 2]
y_pred = [0, 0, 2, 2, 1]

target_names = ['class 0', 'class 1', 'class 2']
print(classification_report(y_true, y_pred, target_names=target_names))
```

```
"""
              precision    recall  f1-score   support

     class 0       0.50      1.00      0.67         1
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.67      0.80         3

    accuracy                           0.60         5
   macro avg       0.50      0.56      0.49         5
weighted avg       0.70      0.60      0.61         5
"""
```

5.6.5. Obtain Reproducible Optimizations Results in Optuna

Optuna is a powerful hyperparameter optimization framework that supports many machine learning frameworks, including TensorFlow, PyTorch, and XGBoost.

But you need to be careful with reproducible results for hyperparameter tuning.tuple

To achieve reproducible results, you need to set the seed for your Sampler.

Below you can see how it is done for `TPESampler`.

```python
import optuna
from optuna.samplers import TPESampler

def objective(trial):
    ...

sampler = TPESampler(seed=42)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)
```

5.6.6. Find bad labels with `doubtlab`

Do you want to find bad labels in your data?

Try `doubtlab` for Python.

With `doubtlab`, you can define reasons to doubt your labels and take a closer look.

Skip to main content

Reasons to doubt your labels can be for example:

- **ProbaReason**: When the confidence values are low for any label
- **WrongPredictionReason**: When a model cannot predict the listed label
- **DisagreeReason**: When two models disagree on a prediction.
- **RelativeDifferenceReason**: When the relative difference between label and prediction is too high

So, identify your noisy labels and fix them.

```
!pip install doubtlab
```

```
from doubtlab.ensemble import DoubtEnsemble
from doubtlab.reason import ProbaReason, WrongPredictionReason
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

X, y = load_iris(return_X_y=True)
model = LogisticRegression()
model.fit(X, y)

# Define reasons to check
reasons = {
    'proba': ProbaReason(model=model),
    'wrong_pred': WrongPredictionReason(model=model),
}

# Pass reasons to DoubtLab instance
doubt = DoubtEnsemble(**reasons)

# Returns DataFrame with reasoning
predicates = doubt.get_predicates(X, y)
```

5.6.7. Get notified when your model is finished with training

Never stare at your screen, waiting for your model to finish training.

Try `knockknock` for Python.

`knockknock` is a library that notifies you when your training is finished.

Skip to main content

Currently, you can get a notification through 12 different channels like:

- Email
- Slack
- Telegram
- Discord
- MS Teams

Use it for your future model training and don't stick to your screen.

```
!pip install knockknock
```

```
from knockknock import email_sender

@email_sender(recipient_emails=["coolmail@python.com", "2coolmail@python.com"], send
def train_model(model, X, y):
    model.fit(X, y)
```

5.6.8. Get Model Summary in PyTorch with `torchinfo`

Do you want a Model summary in PyTorch?

Like in Keras with `model.summary()`?

Use `torchinfo`.

With `torchinfo`, you can get a model summary as you know it from Keras.

Just add one line of code.

```
!pip install torchinfo
```

```python
import torch
from torchinfo import summary

class MyModel(torch.nn.Module)
  ...

model = MyModel()

BATCH_SIZE = 16
summary(model, input_size=(BATCH_SIZE, 1, 28, 28))
```

```
'''
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      [16, 10]                  --
├─Sequential: 1-1                        [16, 4, 7, 7]             --
│    └─Conv2d: 2-1                       [16, 4, 28, 28]           40
│    └─BatchNorm2d: 2-2                  [16, 4, 28, 28]           8
│    └─ReLU: 2-3                         [16, 4, 28, 28]           --
│    └─MaxPool2d: 2-4                    [16, 4, 14, 14]           --
│    └─Conv2d: 2-5                       [16, 4, 14, 14]           148
│    └─BatchNorm2d: 2-6                  [16, 4, 14, 14]           8
│    └─ReLU: 2-7                         [16, 4, 14, 14]           --
│    └─MaxPool2d: 2-8                    [16, 4, 7, 7]             --
├─Sequential: 1-2                        [16, 10]                  --
│    └─Linear: 2-9                       [16, 10]                  1,970
==========================================================================================
Total params: 2,174
Trainable params: 2,174
Non-trainable params: 0
Total mult-adds (M): 1.00
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 1.00
Params size (MB): 0.01
Estimated Total Size (MB): 1.06
==========================================================================================
'''
```

5.6.9. Boost scikit-learns performance with Intel Extension

Scikit-learn is one of the most popular ML packages for Python.

But, to be honest, their algorithms are not the fastest ones.

Skip to main content

With Intel's Extension for scikit-learn, `scikit-learn-intelex`. you can speed up training time for some favourite algorithms like:

- Support Vector Classifier/Regressor
- Random Forest Classifier/Regressor
- LASSO
- DBSCAN

Just add two lines of code.

```
!pip install scikit-learn-intelex
```

```
from sklearnex import patch_sklearn
patch_sklearn()

from sklearn.svm import SVR
from sklearn.datasets import make_regression

X, y = make_regression(
n_samples=100000,
n_features=10,
noise=0.5)

svr = SVR()

svr.fit(X, y)
```

5.6.10. Incorportate Domain Knowledge into XGBoost with Feature Interaction Constraints

Want to incorporate your domain knowledge into **XGBoost**?

Try using **Feature Interaction Constraints**.

Feature Interaction Constraints allow you to control which features are allowed to interact with each other and which are not while building the trees.

For example, the constraint [0, 1] means that Feature_0 and Feature_1 are allowed to interact with each other but with no other variable. Similarly, [3, 5, 9] means that Feature_3, Feature_5, and

Skip to main content

With this in mind, you can define feature interaction constraints:

- Based on domain knowledge, when you know that some features interactions will lead to better results
- Based on regulatory constraints in your industry/company where some features can not interact with each other.

```
import xgboost as xgb

X, y = ...

dmatrix = xgb.DMatrix(X, label=y)

params = {
    "objective": "reg:squarederror",
    "eval_metric": "rmse",
    "interaction_constraints": [[0,2 ], [1, 3, 4]]
}

model_with_constraints = xgb.train(params, dmatrix)
```

5.6.11. Powerful AutoML with `FLAML`

Do you always hear about AutoML?

And want to try it out?

Use `FLAML` for Python.

`FLAML` (Fast and Lightweight AutoML) is an AutoML package developed by Microsoft.

It can do Model Selection, Hyperparameter tuning, and Feature Engineering automatically.

Thus, it removes the pain of choosing the best model and parameters so that you can focus more on your data.

Per default, its estimator list contains only tree-based models like XGBoost, CatBoost, and LightGBM. But you can also add custom models.

A powerful library!

Skip to main content

```
!pip install flaml
```

```
from flaml import AutoML
automl = AutoML()
automl.fit(X_train, y_train, task="classification")
```

5.6.12. Aspect-based Seniment Analysis with `PyABSA`

Traditional sentiment analysis focuses on determining the overall sentiment of a piece of text.

For example, the sentence :

“The food was bad and the staff was rude”

would output only a negative sentiment.

But, what if I want to extract, which aspects have a negative or positive sentiment?

That’s the responsibility of aspect-based sentiment analysis.

It aims to identify and extract the sentiment expressed towards specific aspects of a text.

For the sentence:

”The battery life is excellent but the camera quality is bad.”

a model’s output would be:

- Battery life: positive
- Camera quality: negative

With aspect-based sentiment analysis, you can understand the opinions and feelings expressed about specific aspects.

To do that in Python, use the package `PyABSA`.

It contains pre-trained models with an easy-to-use API for aspect-term extraction and sentiment

Skip to main content

`PyABSA` can be used for a variety of applications, such as:

- Customer feedback analysis
- Product reviews analysis
- Social media monitoring

```
!pip install pyabsa==1.16.27
```

```
from pyabsa import ATEPCCheckpointManager

extractor = ATEPCCheckpointManager.get_aspect_extractor(
                  checkpoint="multilingual",
                  auto_device=False
)

example = ["Location and food were excellent but stuff was very unfriendly."]
result = extractor.extract_aspect(inference_source=example, pred_sentiment=True)

print(result)
```

5.6.13. Use XGBoost for Random Forests

Are you still using Random Forests from sklearn?

XGBoost implements Random Forests too, and much faster than sklearn.

```
from xgboost import XGBRFRegressor

xgbrf = XGBRFRegressor(n_estimators=100)

X = np.random.rand(100000, 10)
y = np.random.rand(100000)

xgbrf.fit(X, y)
```

5.6.14. Identify problematic images with `cleanvision`

Your Deep Learning Model doesn't perform?

Skip to main content

With `cleanvision`, you can detect issues in image data.

`cleanvision` is a relatively new data-centric AI package to find problems in your image dataset.

It can detect issues like:

- Exact or Near Duplicates
- Blurry Images
- Odd Aspect Ratios
- Irregularly Dark/Light images
- Images lacking content

A good first step to try before applying crazy Vision Transformers.

```
!wget - nc 'https://cleanlab-public.s3.amazonaws.com/CleanVision/image_files.zip'
```

```
!unzip -q image_files.zip
```

```
!pip install cleanvision
```

```
from cleanvision.imagelab import Imagelab

# Path to your dataset, you can specify your own dataset path
dataset_path = "./image_files/"

# Initialize imagelab with your dataset
imagelab = Imagelab(data_path=dataset_path)

# Find issues
imagelab.find_issues()
```

```
# Get summary of issues with prevalence per issue
imagelab.issue_summary
```

```
# Visualize Top examples for blurry images
imagelab.visualize(issue_types=['blurry'])
```

Skip to main content

5.6.15. Select the optimal Regularization Parameter

How do you choose your Regularization Parameter?

Your model's complexity decreases with a higher Regularization Parameter (Alpha).

It shouldn't be too high or too low.

Yellowbrick's `AlphaSelection` can help you to find the best Alpha.

It takes your model and visualizes the Alpha/Error curve so you can see how the model's error responds to different alpha values.

Below you can see how to do it with scikit-learn's LassoCV.

```
import numpy as np
from sklearn.linear_model import LassoCV
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import AlphaSelection

X, y = load_concrete()

# Create a list of alphas to cross-validate against
alphas = np.linspace(0, 10, 30)

model = LassoCV(alphas=alphas)
visualizer = AlphaSelection(model)
visualizer.fit(X, y)
visualizer.show()
```

5.6.16. Decision Forests in TensorFlow

Did you know there are Decision Forests from TensorFlow?

`tensorflow_decision_forests` implements decision forest models like Random Forest or GBDT for classification, regression, and ranking.

```
!pip install tensorflow_decision_forests
```

Skip to main content

```
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_decision_forests as tfdf
```

```
dataset_path = tf.keras.utils.get_file(
      "adult.csv",
      "https://raw.githubusercontent.com/google/yggdrasil-decision-forests/"
      "main/yggdrasil_decision_forests/test_data/dataset/adult.csv")

dataset_df = pd.read_csv(dataset_path)
test_indices = np.random.rand(len(dataset_df)) < 0.30
test_ds_pd = dataset_df[test_indices]
train_ds_pd = dataset_df[~test_indices]


train_ds = tfdf.keras.pd_dataframe_to_tf_dataset(train_ds_pd, label="income")
test_ds = tfdf.keras.pd_dataframe_to_tf_dataset(test_ds_pd, label="income")
```

```
model = tfdf.keras.GradientBoostedTreesModel(verbose=2)
model.fit(train_ds)

print(model.summary())
```

5.6.17. AutoML with `AutoGluon`

Do you always hear about AutoML?

And want to try it out?

Use `AutoGluon` for Python.

`AutoGluon` is a Python package from AWS.

It lets you perform AutoML on:

- Tabular Data (Classification, Regression)
- Time Series Data
- Multimodal Data (Images + Text + Tabular)

AutoGluon also offers utilities for EDA, like:

- Detecting Covariate Shift
- Target Variable Analysis
- Feature Interaction Charts

See below for a quickstart for tabular data.

```
!pip install autogluon
```

```
from autogluon.tabular import TabularDataset, TabularPredictor

train_data = TabularDataset('https://autogluon.s3.amazonaws.com/datasets/Inc/train.c
test_data = TabularDataset('https://autogluon.s3.amazonaws.com/datasets/Inc/test.csv

predictor = TabularPredictor(label='class').fit(train_data, time_limit=240)
predictor.leaderboard(test_data)
```

5.6.18. Visualize Keras Models with `visualkeras`

Do you want some cool visualization for your Deep Learning Models?

Try `visualkeras`.

`visualkeras` visualizes your Keras models (as an alternative to model.summary())

```
!pip install visualkeras
```

```
import visualkeras

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

visualkeras.layered_view(model, legend=True, to_file='output.png').show()
```

5.6.19. Perform Multilabel Stratified KFold with `iterative-stratification`

When doing Cross-Validation for classification,

StratifiedKFold from scikit-learn is a common choice.

Stratification aims to guarantee that every fold represents all strata of the data.

But, scikit-learn doesn't support stratifying multilabel data.

For this use case, try the `iterative-stratification` package.

It offers implementations for stratyfing multilabel data in different ways.

See below how we can use MultilabelStratifiedKFold.

```
!pip install iterative-stratification
```

```
from iterstrat.ml_stratifiers import MultilabelStratifiedKFold
import numpy as np

X = np.array([[1,2], [3,4], [1,2], [3,4], [1,2], [3,4], [1,2], [3,4]])
y = np.array([[0,0], [0,0], [0,1], [0,1], [1,1], [1,1], [1,0], [1,0]])

mskf = MultilabelStratifiedKFold(n_splits=2, shuffle=True, random_state=0)

for train_index, test_index in mskf.split(X, y):
   print("TRAIN:", train_index, "TEST:", test_index)
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]
```

5.6.20. Interpret your Model with `Shapash`

Nobody cares about your SOTA ML Model if nobody can't understand the predictions.

Therefore, interpretability of ML Models is a crucial point in industry cases.

To overcome this hurdle, use `Shapash` for Python.

`Shapash` offers several types of interpretability methods to understand your model's predictions like:

- Feature Importance
- Feature Contribution
- LIME
- SHAP

It comes also with an intuitive GUI to interact with.

Check it out! Link is in the comments section.

```
!pip install shapash
```

```
import pandas as pd
from category_encoders import OrdinalEncoder
from lightgbm import LGBMRegressor
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesRegressor

from shapash.data.data_loader import data_loading
house_df, house_dict = data_loading('house_prices')

y_df=house_df['SalePrice'].to_frame()
X_df=house_df[house_df.columns.difference(['SalePrice'])]

from category_encoders import OrdinalEncoder

categorical_features = [col for col in X_df.columns if X_df[col].dtype == 'object']

encoder = OrdinalEncoder(
    cols=categorical_features,
    handle_unknown='ignore',
    return_df=True).fit(X_df)

X_df=encoder.transform(X_df)

Xtrain, Xtest, ytrain, ytest = train_test_split(X_df, y_df, train_size=0.75, random_

regressor = LGBMRegressor(n_estimators=100).fit(Xtrain,ytrain)

from shapash import SmartExplainer

xpl = SmartExplainer(
    model=regressor,
    preprocessing=encoder,
    features_dict=house_dict
)
```

```
xpl.compile(x=Xtest,
            y_target=ytest
           )
```

```
app = xpl.run_app(title_story='House Prices', port=8020)
```

5.6.21. Validate Your Model and Data with `Deepchecks`

Validating your Model and Data is crucial in ML.

Skip to main content

Not testing them will cause huge problems in production.

To change that, use `deepchecks`.

`deepchecks` is an open-source solution which offers a suite for detailed validation methods.

It will calculate and visualize a bunch of things like:

- Train/Test Performance
- Predictive Power Score
- Feature Drift
- Label Drift
- Weak Segments for your model

A powerful tool to consider for testing your models and datasets.

```
!pip install deepchecks
```

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from deepchecks.tabular.datasets.classification import iris
from deepchecks.tabular import Dataset
from deepchecks.tabular.suites import full_suite

# Load Data
iris_df = iris.load_data(data_format='Dataframe', as_train_test=False)
label_col = 'target'
df_train, df_test = train_test_split(iris_df, stratify=iris_df[label_col], random_st

# Train Model
rf_clf = RandomForestClassifier(random_state=0)
rf_clf.fit(df_train.drop(label_col, axis=1), df_train[label_col])


ds_train = Dataset(df_train, label=label_col, cat_features=[])
ds_test =  Dataset(df_test,  label=label_col, cat_features=[])

suite = full_suite()

suite.run(train_dataset=ds_train, test_dataset=ds_test, model=rf_clf)
```

Skip to main content

5.6.22. Visualize high-performance Features with `Optuna`

Optuna released a new feature for detecting high-performing parameters.

Its `plot_rank()` function visualizes different parameters, with individual points representing individual trials.

Since the plot is interactive, you can also hover over it and dive deeper into analysing your hyperparameter optimization.

```
from sklearn.ensemble import RandomForestClassifier

import optuna


def objective(trial):
    clf = RandomForestClassifier(
        n_estimators=50,
        criterion="gini",
        max_depth=trial.suggest_int('Mdpth', 2, 32, log=True),
        min_samples_split=trial.suggest_int('mspl', 2, 32, log=True),
        min_samples_leaf=trial.suggest_int('mlfs', 1, 32, log=True),
        min_weight_fraction_leaf=trial.suggest_float('mwfr', 0.0, 0.5),
        max_features=trial.suggest_int("Mfts", 1, 15),
        max_leaf_nodes=trial.suggest_int('Mnods', 4, 100, log=True),
        min_impurity_decrease=trial.suggest_float('mid', 0.0, 0.5),
    )
    clf.fit(X_train, y_train)
    return clf.score(X_test, y_test)

# Optimize
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=30)

# Get parameters sorted by the importance values
importances = optuna.importance.get_param_importances(study)
params_sorted = list(importances.keys())

# Plot
fig = optuna.visualization.plot_rank(study, params=params_sorted[:4])
fig.show()
```

5.6.23. Model Ensembling with `combo`

Looking at the top solutions on Kaggle you will notice one thing:

There is usually some sort of combination of various ML models involved.

With `combo` for Python, you can combine

- Multiple Classifiers
- Multiple Anomaly Detection Models
- Multiple Clustering Models

`combo` also offers multiple combination methods for every category.

```
!pip install combo
```

```
from combo.models.cluster_comb import ClustererEnsemble

estimators = [KMeans(n_clusters=n_clusters),
              MiniBatchKMeans(n_clusters=n_clusters),
              AgglomerativeClustering(n_clusters=n_clusters)]

clf = ClustererEnsemble(estimators, n_clusters=n_clusters)
clf.fit(X)

aligned_labels = clf.aligned_labels_
predicted_labels = clf.labels_
```

5.6.24. Residual Plots with `yellowbrick`

To analyze the variance of the error of your Regression model

Use `ResidualPlot` from `yellowbrick`.

With Residual Plots, you can see how well-fitted your model is.

If the data points exhibit a random distribution along the horizontal axis, a linear regression model is

See below how you can easily implement that with `yellowbrick`.

```
!pip install yellowbrick
```

```
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import ResidualsPlot

model = Lasso()
visualizer = ResidualsPlot(model)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```

5.6.25. Powerful and Distributed Hyperparameter Optimization with `ray.tune`

Do you need hyperparameter tuning on steroids?

Try `tune` from `ray`.

`tune` performs distributed hyperparameter tuning with multi-GPU and multi-node support, utilizing all the hardware you have.

It supports the most popular ML libraries and integrates many other common hyperparameter optimization tools like Optuna or Hyperopt.

```
!pip install "ray[tune]"
```

```
# !pip install "ray[tune]"
import sklearn.datasets
import sklearn.metrics
import sklearn.datasets
import sklearn.metrics
import xgboost as xgb
from ray import train, tune
from sklearn.model_selection import train_test_split


def train_breast_cancer(config):
    data, labels = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, test_x, train_y, test_y = train_test_split(data, labels, test_size=0.25
    train_set = xgb.DMatrix(train_x, label=train_y)
    test_set = xgb.DMatrix(test_x, label=test_y)
    results = {}
    xgb.train(
        config,
        train_set,
        evals=[(test_set, "eval")],
        evals_result=results,
        verbose_eval=False,
    )
    accuracy = 1.0 - results["eval"]["error"][-1]
    train.report({"mean_accuracy": accuracy, "done": True})


config = {
    "objective": "binary:logistic",
    "eval_metric": ["logloss", "error"],
    "min_child_weight": tune.choice([1, 2, 3]),
    "subsample": tune.uniform(0.5, 1.0),
}

tuner = tune.Tuner(
    train_breast_cancer,
    tune_config=tune.TuneConfig(
        num_samples=10,
    ),
    param_space=config,
)
results = tuner.fit()
print(results.get_best_result(metric="mean_accuracy", mode="max").config)
```

5.6.26. Use PyTorch with scikit-learn API with `skorch`

PyTorch and scikit-learn are one of the most popular libraries for ML/DL.

Try `skorch`!

`skorch` is a high-level library for PyTorch that provides a scikit-learn-compatible neural network module.

It allows you to use the simple scikit-learn interface for PyTorch.

Therefore you can integrate PyTorch models into scikit-learn workflows.

See below for an example.

```
!pip install skorch
```

```
from torch import nn
from skorch import NeuralNetClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

class MyModule(nn.Module):
    def __init__(self, num_units=10, nonlin=nn.ReLU()):
        super().__init__()

        self.dense = nn.Linear(20, num_units)
        self.nonlin = nonlin
        self.output = nn.Linear(num_units, 2)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, X, **kwargs):
        X = self.nonlin(self.dense(X))
        X = self.dropout(X)
        X = self.softmax(self.output(X))
        return X

net = NeuralNetClassifier(
    MyModule,
    max_epochs=10,
    lr=0.1,
    iterator_train__shuffle=True,
)

pipe = Pipeline([
    ('scale', StandardScaler()),
    ('net', net),
])

pipe.fit(X, y)
```

Skip to main content

5.6.27. Online ML with `river`

Do you want ML models that learn on-the-fly from massive datasets?

Try `river`.

`river` is a library for online machine learning.

You can continuously update your model with streaming data without using the full dataset for training again.

It provides online implementations for many algorithms like KNN, Tree-based models and Recommender systems.

```
!pip install river
```

```
from river import compose
from river import linear_model
from river import metrics
from river import preprocessing
from river import datasets

dataset = datasets.Phishing()

model = compose.Pipeline(
    preprocessing.StandardScaler(),
    linear_model.LogisticRegression()
)

metric = metrics.Accuracy()

for x, y in dataset:
    y_pred = model.predict_one(x)
    metric.update(y, y_pred)
    model.learn_one(x, y)
```

5.6.28. SOTA Computer Vision Models with `timm`

Do you want to use SOTA computer vision models?

Skip to main content

Try `timm`.

`timm` (PyTorch Image Models) is a library which contains multiple computer vision models, layers, optimizers, etc.

It provides models like Vision Transformer, MobileNet, Swin Transformer, ConvNeXt, DenseNet, and more.

You just have to define the name of the model and if you want to have the pretrained weights of it.

```
!pip install timm
```

```
import torch
import timm

print(timm.list_models())

model = timm.create_model('densenet121', pretrained=True)
output = model(torch.randn(2, 3, 224, 224))
```

5.6.29. Generate Guaranteed Prediction Intervals and Sets with `MAPIE`

For quantifying uncertainties of your models, use MAPIE.

`MAPIE` (Model Agnostic Prediction Interval Estimator) takes your sklearn-/tensorflow-/pytorch-compatible model and generate prediction intervals or sets with guaranteed coverage.

```
!pip install mapie
```

```
from mapie.regression import MapieRegressor
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=500, n_features=1, noise=20, random_state=59)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
regressor = LinearRegression()

mapie_regressor = MapieRegressor(regressor)
mapie_regressor.fit(X_train, y_train)

alpha = [0.05, 0.20]
y_pred, y_pis = mapie_regressor.predict(X_test, alpha=alpha)
```

5.6.30. Extra Components For scikit-learn with `scikit-lego`

scikit-learn is one of the most popular ML libraries.

While it's easy to write custom components, it would be nice to have all of them in a single place.

`scikit-lego` is such a library which contains many custom components like:

- `DebugPipeline`, which adds debug information to pipelines
- `ImbalancedLinearRegression` to punish over-/underestimation of a model
- `add_lags` to add lag values to a DataFrame
- `ZeroInflatedRegressor` which predicts zero or applies a regression based on a classifier

and many more!

```
!pip install scikit-lego
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklego.preprocessing import RandomAdder
from sklego.mixture import GMMClassifier

...

pipeline = Pipeline([
    ("scale", StandardScaler()),
    ("random_noise", RandomAdder()),
    ("model", GMMClassifier())
])

...
```

5.6.31. Quantize your Models with `torchao`

Quantizing your Deep Learning models was never easier.

With `torchao`, you can quantize and sparsify your models with 1 line of code.

If you are unsure which method to use, you can even use the autoquant method to quantize your layers automatically.

```
!pip install torchao
```

```
import torchao

model = torchao.autoquant(torch.compile(model, mode='max-autotune'))
```

5.7. Outlier Detection

5.7.1. Ensembling for Outlier Detection

Due to its unsupervised nature, outlier detection methods often suffer from model instability.

So, why not combine various models?

Skip to main content

Try `PyOD`!

`PyOD` is an easy-to-use library for outlier detection.

It includes more than 30 algorithms like density-based methods or ensembles.

`PyOD` also supports combining multiple methods like

- Average of scores
- Maximization of scores
- Average of Maximum of scores
- Maximum of Average of scores
- Majority Vote

To combine multiple models in Python, consider the example below.

- We define 3 outlier detectors.
- We calculate the labels for every detector (0=inliner, 1=outlier).
- We use `majority_vote()` method to calculate the highest-voted label for each sample.

```
!pip install pyod
```

```
import numpy as np
from pyod.models.combination import majority_vote
from pyod.models.knn import KNN
from pyod.models.abod import ABOD
from pyod.models.iforest import IForest
from pyod.utils.data import generate_data

X, _= generate_data(train_only=True)

models = [KNN(), ABOD(), IForest()]
n_models = len(models)

labels = np.zeros([X.shape[0], n_models])

for i in range(n_models):
    model = models[i]

    model.fit(X)

    labels[:, i] = model.labels_

majority_vote(labels)
```

5.7.2. Robust Outlier Detection with `puncc`

Outlier Detection is notoriously hard.

But it doesn't have to.

`puncc` offers outlier detection, powered by Conformal Prediction, where the detection threshold will be calibrated.

So, false alarms are reduced.

```
!pip install puncc
```

```
from sklearn.ensemble import IsolationForest
from deel.puncc.anomaly_detection import SplitCAD
from deel.puncc.api.prediction import BasePredictor

# We need to redefine the predict to output the nonconformity scores.
class ADPredictor(BasePredictor):
    def predict(self, X):
        return -self.model.score_samples(X)

# Wrap Isolation Forest in a predictor
if_predictor = ADPredictor(IsolationForest())

# Instantiate CAD on top of IF predictor
if_cad = SplitCAD(if_predictor, train=True)


if_cad.fit(z=dataset, fit_ratio=0.7)

# Maximum false detection rate
alpha = 0.01

results = if_cad.predict(new_data, alpha=alpha)
```

5.8. Time Series

5.8.1. Check Seasonality automatically with `darts`

Seasonality describes a pattern that repeats regularly over time.

Identifying and understanding the seasonality in time series can boost the performance of your model.

But you don't have to find the seasonality effect and period by yourself.

Instead, you can use `check_seasonality()` from `darts` in Python.

It will check if the time series is seasonal and returns also the period, which is inferred from the Auto-correlation Function.

In the example below, it will return a seasonal period of 12 (Air Passenger Dataset has a monthly frequency).

Skip to main content

```
!pip install darts
```

```
from darts.utils.statistics import check_seasonality
from darts.datasets import AirPassengersDataset

ts = AirPassangersDataset().load()

is_seasonal, period = check_seasonality(ts)
```

5.8.2. Cross-validation for Time Series Data with `TimeSeriesSplit`

How to do Cross-Validation with Time Series?

Using standard K-Fold Cross-Validation will not work.

In this case, you would simply partition the data into k folds, and then train and evaluate the model k times, each time using a different fold as the test set and the rest of the data as the training set.

But, this can lead to issues because the model will be trained on data that is both before and after the test data.

This can result in overfitting or biased estimates of model performance

Instead, use `TimeSeriesSplit` from scikit-learn.

`TimeSeriesSplit` ensures that the model is only trained on the past values and tested on future data.

This gives you a more accurate and less biased assessment of the model's performance.

```
from sklearn.model_selection import TimeSeriesSplit, cross_validate
from sklearn.ensemble import GradientBoostingRegressor

X, y = ...
model = GradientBoostingRegressor()

ts_cv = TimeSeriesSplit(n_splits=3)

scores = cross_validate(model, X, y, cv=ts_cv, scoring='neg_mean_squared_error')
```

Skip to main content

5.8.3. More Cross-Validation with `tscv`

How to do Cross-Validation with Time Series?

Using standard K-Fold Cross-Validation will not work.

In this case, you would simply partition the data into k folds, and then train and evaluate the model k times, each time using a different fold as the test set and the rest of the data as the training set.

But, this can lead to issues because the model will be trained on data that is both before and after the test data.

This can result in overfitting or biased estimates of model performance.

Instead, use `tscv` package for Python.

`tscv` offers methods for correct splitting of your data with 3 classes implemented:

- `GapLeavePOut`
- `GapKFold`
- `GapRollForward`

This gives you a more accurate and less biased assessment of the model's performance.

```
!pip install tscv
```

```
from tscv import GapRollForward
cv = GapRollForward(min_train_size=3, gap_size=1, max_test_size=2)
for train, test in cv.split(range(10)):
    print("train:", train, "test:", test)
```

5.8.4. Time Series Forecasting with Machine Learning with `mlforecast`

Do you want to perform powerful time series forecasting?

Skip to main content

Try `mlforecast` by Nixtla.

`mlforecast` lets you run Machine Learning models for time series forecasting, even on remote clusters like Ray or Spark.

Feature Engineering, support for exogenous variables, and probabilistic forecasting are also included.

```
!pip install mlforecast
```

```
import lightgbm as lgb

from mlforecast import MLForecast
from sklearn.linear_model import LinearRegression

mlf = MLForecast(
    models = [LinearRegression(), lgb.LGBMRegressor()],
    lags=[1, 12],
    freq = 'M'
)
mlf.fit(df)
mlf.predict(12)
```

5.8.5. Lightning Fast Time Series Forecasting with `statsforecast`

Do you want to perform lightning fast time series forecasting?

Try `statsforecast` by Nixtla.

`statsforecast` lets you run statistical models on your time series data.

It's up to 20x faster than existing libraries like pmdarima and statsmodels.

```
!pip install statsforecast
```

```
from statsforecast import StatsForecast
from statsforecast.models import AutoARIMA
from statsforecast.utils import AirPassengersDF

df = AirPassengersDF
sf = StatsForecast(
    models = [AutoARIMA(season_length = 12)],
    freq = 'M'
)

sf.fit(df)
sf.predict(h=12, level=[95])
```

5.8.6. Time Series with Polars Backend with `functime`

Fast time-series forecasting with `functime`.

`functime` is a Python library for time series forecasting and feature extraction, built with Polars.

Since it uses lazy Polars dataframes, `functime` speeds up forecasting and feature engineering.

Backtesting, cross-validation splitters and metrics are included too.

It even comes with a LLM agent to analyze and describe your forecasts.

Check it out!

```
!pip install functime
```

```
import polars as pl
from functime.cross_validation import train_test_split
from functime.forecasting import linear_model
from functime.metrics import mase

y_train, y_test = y.pipe(train_test_split(test_size=3))

forecaster = linear_model(freq="1mo", lags=24)
forecaster.fit(y=y_train)
y_pred = forecaster.predict(fh=3)

y_pred = linear_model(freq="1mo", lags=24)(y=y_train, fh=3)
```

5.8.7. Time Series Forecasting with Deep Learning with `neuralforecast`

Do you want to perform powerful time series forecasting?

Try `neuralforecast` by nixtla.

`neuralforecast` lets you run Deep Learning models for time series forecasting with models like N-BEATS or N-HiTS.

Support for exogenous variables and probabilistic forecasting are also included.

Check the example below!

```
!pip install neuralforecast
```

```
import pandas as pd

from neuralforecast import NeuralForecast
from neuralforecast.models import NBEATS, NHITS
from neuralforecast.utils import AirPassengersDF

Y_df = AirPassengersDF
Y_train_df = Y_df[Y_df.ds<='1959-12-31']
Y_test_df = Y_df[Y_df.ds>'1959-12-31']

horizon = 12
models = [NBEATS(input_size=2 * horizon, h=horizon, max_steps=50),
          NHITS(input_size=2 * horizon, h=horizon, max_steps=50)]

nf = NeuralForecast(models=models, freq='M')
nf.fit(df=Y_train_df)
Y_hat_df = nf.predict().reset_index()
```

5.8.8. Efficient Preprocessing and Feature Engineering with `temporian`

`temporian` is a Python library for preprocessing and feature engineering temporal data to feed into

Skip to main content

It handles various types of temporal data like single- and multivariate data or flat- and multi-index data.

```
!pip install temporian
```

```
import temporian as tp

sales = tp.from_csv("sales.csv")

sales_per_store = sales.add_index("store")

days = sales_per_store.tick_calendar(hour=22)
work_days = (days.calendar_day_of_week() <= 5).filter()

daily_revenue = sales_per_store["revenue"].moving_sum(
                    tp.duration.days(1),
                    sampling=work_days)
```

5.8.9. Change Point Detection with `ruptures`

Change point detection was never easier in Python with `ruptures``

`ruptures` is a library which provides methods for detecting and displaying off-line change points.

It offers multiple exact and approximation detection methods.

```
!pip install ruptures
```

```
import matplotlib.pyplot as plt
import ruptures as rpt

# Generate signal
n_samples, dim, sigma = 1000, 3, 4
n_breakpoints = 4
signal, bkps = rpt.pw_constant(n_samples, dim, n_breakpoints, noise_std=sigma)

# Detection
algo = rpt.Pelt(model="rbf").fit(signal)
result = algo.predict(pen=10)

# Display
rpt.display(signal, bkps, result)
plt.show()
```

5.8.10. Probabilistic Machine Learning with `skpro`

Use supervised probabilistic prediction like a pro with `skpro`.

`skpro` is a scikit-learn-like library for probabilistic predictions and evaluations.

It supports tabular regressors, survival prediction, and reductions to turn scikit-learn regressors into probabilistic `skpro` regressors.

```
!pip install skpro
```

```
from sklearn.datasets import load_diabetes
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

from skpro.regression.residual import ResidualDouble

X, y = load_diabetes(return_X_y=True, as_frame=True)
X_train, X_new, y_train, _ = train_test_split(X, y)

reg_mean = RandomForestRegressor()
reg_resid = LinearRegression()
reg_proba = ResidualDouble(reg_mean, reg_resid)

reg_proba.fit(X_train, y_train)

y_pred_proba = reg_proba.predict_proba(X_new)

y_pred_interval = reg_proba.predict_interval(X_new, coverage=0.9)

y_pred_quantiles = reg_proba.predict_quantiles(X_new, alpha=[0.05, 0.5, 0.95])

y_pred_var = reg_proba.predict_var(X_new)

y_pred_mean = reg_proba.predict(X_new)
```

5.8.11. Evaluating Forecasts with `fev`

Time Series Forecasting without evaluation is guessing, not knowing.

Make your life easier and use **fev**.

fev is a new Python library aiming to benchmark forecasting models easily.

As a wrapper on top of Huggingface Datasets, it is very easy to define custom forecasting benchmarks.

It supports point and even probabilistic forecasts which is crucial in today's world.

```
!pip install fev
```

Skip to main content

```
import fev

# Create Task
task = fev.Task(
    dataset_path="autogluon/chronos_datasets",
    dataset_config="monash_kdd_cup_2018",
    horizon=12,
)
# Load data
past_data, future_data = task.get_input_data()

def naive_forecast(y: list, horizon: int) -> list:
    return [y[-1] for _ in range(horizon)]

# Make predictions
predictions = []
for ts in past_data:
    predictions.append(
        {"predictions": naive_forecast(y=ts[task.target_column], horizon=task.horiz
    )


# Evaluate
task.evaluation_summary(predictions, model_name="naive")
```

5.9. Preprocessing

5.9.1. Clean your text data with `clean-text`

Content on the Web and in Social Media is never clean.

`clean-text` does the Preprocessing for you.

You can specify, if and how you want to clean your texts.

```
!pip install clean-text[gpl]
```

```
from cleantext import clean

text = '''
       If you want to talk, send me an email: testmail@outlook.com,
       call me +71112392 or visit my website: https://testurl.com.
       Calling me is not free, It'\\u2018s\\u2019 costing 0.40$ per
       minute.
       '''

clean(text,
    fix_unicode=True,               # fix various unicode errors
    to_ascii=True,                  # transliterate to closest ASCII representation
    lower=True,                     # lowercase text
    no_urls=True,                   # replace all URLs with a special token
    no_emails=True,                 # replace all email addresses with a special toke
    no_phone_numbers=True,          # replace all phone numbers with a special token
    no_numbers=True,                # replace all numbers with a special token
    no_digits=True,                 # replace all digits with a special token
    no_currency_symbols=True,       # replace all currency symbols with a special tol
    no_punct=True,                  # remove punctuations
    lang="en"                       # set to 'de' for German special handling
)
```

5.9.2. Detect and Fix your Data Quality Issues

Do you want to detect data quality issues?

Try `pandas_dq`.

`pandas_dq` is a relatively new library, focussing on detecting data quality issues and fixing them automatically like:

- Zero-Variance Columns
- Rare Categories
- Highly correlated Features
- Skewed Distributions

```
!pip install pandas_dq -q
```

```
import pandas as pd
import numpy as np
from pandas_dq import dq_report, Fix_DQ
from sklearn.datasets import load_iris
```

```
data = load_iris()
```

```
data = pd.DataFrame(data= np.c_[data['data'], data['target']],
                    columns= data['feature_names'] + ['target'])
```

```
dq_report(data, verbose=1)
```

```
fdq = Fix_DQ()
data_transformed = fdq.fit_transform(data)
```

5.9.3. Convert Natural Language Numbers into its Numerical Representation

If you want to convert natural language numbers into numerical values, try numerizer.

`numerizer` is a Python library for converting numbers in texts to their corresponding numerical values.

`numerizer` supports a wide range of numeric formats, including whole numbers, decimals, percentages and currencies.

Note: Since version 0.2, `numerizer` is available as a SpaCy extension.

```
!pip install numerizer
```

```
from numerizer import numerize

text_1 = "Twenty five dollars"
text_2 = "Two hundred and fourty three thousand four hundred and twenty one"
text_3 = "platform nine and three quarters"


num_1 = numerize(text_1)
num_2 = numerize(text_2)
num_3 = numerize(text_3)

print(num_1) # Output: 25 dollars
print(num_2) # Output: 243421
print(num_3) # Output: platform 9.75
```

5.9.4. Make your Numbers and dates human-friendly

Looking to make your numbers human-friendly?

Try `humanize`.

`humanize` formats your numbers and dates in a way that is intuitive to understand.

It provides various functionalities like:

- Convert large integers of file sizes
- Convert floats to fractions
- Convert dates into a human-understandable format
- Make big integers more readable

```
!pip install humanize
```

```
import humanize

# Convert bytes to human readable format
humanize.naturalsize(1024000) # Output: 1.0 MB

# Convert a number to its word equivalent
humanize.intword(123500000) # Output: 123.5 million

# Convert a float to its fractional equivalent
humanize.fractional(0.9) # Output: 9/10

# Convert seconds to a readable format
import datetime as dt
humanize.naturaldelta(dt.timedelta(seconds = 1200)) # Output: 20 minutes
```

5.9.5. Cleaner Pipeline definition in Scikit-Learn

When you build Pipelines in scikit-learn,

use `make_pipeline` instead of the Pipeline class.

The Pipeline class can be really long for more complex pipelines.

`make_pipeline` makes your pipeline definition short and elegant.

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(KNNImputer(), RobustScaler())
cat_pipeline = make_pipeline(SimpleImputer("most_frequent"))
```

5.9.6. Select Columns for your Pipeline easily

If you want a convenient way to select columns for your Scikit-learn pipelines

Use `make_column_selector`.

You can even provide complex regex patterns to select the columns you want.

Afterward, you can use the result in your Pipelines easily.

Skip to main content

```
from sklearn.compose import make_column_selector

# Will only select columns with 'Feature' in its name
columns_with_feature = make_column_selector(pattern='Feature')

# Will only select numeric columns
num_columns = make_column_selector(dtype_include="category")
```

5.9.7. Rare Label Encoding with `feature-engine`

How to tackle Rare Labels in your dataset?

Rare labels can cause issues during model training, as they may not have sufficient representation for the model to learn meaningful patterns.

For this problem, use `RareLabelEncoder` from `feature_engine`.

It will convert all rare labels (based on a threshold) to the label “Rare”.

```
!pip install feature_engine
```

```
import pandas as pd
from feature_engine.encoding import RareLabelEncoder

data = ['red', 'blue', 'red', 'green', 'yellow', 'yellow', 'red', "black", "violet",

df = pd.DataFrame({'color': data})

rare_encoder = RareLabelEncoder(tol=0.1, n_categories=5, variables=['color'])

df_encoded = rare_encoder.fit_transform(df)

df
```

6. LLM

6.1. LLM

6.1.1. Compressing Prompts With No Loss with `llmlingua`

Here is how to reduce the costs of working with LLMs.

When working with LLMs, we often encountered problems like exceeding token limits, forgetting context, or paying much more for usage than expected.

Researchers from Microsoft try to solve these problems with `llmlingua`.

`llmlingua` compresses your prompt by taking a trained small LLM to detect unimportant tokens.

They claim to achieve up to 20x compression with no or minimal performance loss.

I tried it out by myself and I noticed no performance loss at all, but I would be cautious for critical applications.

```
!pip install llmlingua
```

```
# !pip install llmlingua

from llmlingua import PromptCompressor

prompt = "<YOUR_PROMPT>"
llm_lingua = PromptCompressor("lgaalves/gpt2-dolly",)

compressed_prompt = llm_lingua.compress_prompt(prompt, instruction="", question="",

# {'compressed_prompt': 'are- that turns into formatting & with like "[]" best it..
# 'origin_tokens': 2430,
# 'compressed_tokens': 261,
# 'ratio': '9.3x',
# 'saving': 'Saving $0.1 in GPT-4.}'
```

6.1.2. One-Function Call to Any LLM with `litellm`

Try `litellm`.

`litellm` is a Python package to call any LLM in a consistent format and to return a consistent output.

You only need to set the API key of the provider and the model name.

It also supports async calls and streaming the models response.

```
!pip install litellm
```

```
import os
from litellm import completion

os.environ["OPENAI_API_KEY"] = "your-api-key"
os.environ["ANTHROPIC_API_KEY"] = "your-api-key"
os.environ['MISTRAL_API_KEY'] = "your-api-key"

messages = [{ "content": "Hello, how are you?","role": "user"}]

# OpenAI
response = completion(model="gpt-3.5-turbo", messages=messages)

# Anthropic
response = completion(model="claude-instant-1", messages=messages)

# Mistral
response = completion(model="mistral/mistral-tiny", messages=messages)
```

6.1.3. Safeguard Your LLMs with `LLMGuard`

Safeguarding your LLMs against unwanting behavior is critical.

`LLMGuard`, a Python package, ensures a safe interaction between the user and LLM.

It checks prompts and outputs for:

- Sensitive Information like credit card number and sanitizes it
- Toxic or harmful language
- Prompt injections

```
!pip install llm-guard
```

```
from openai import OpenAI
from llm_guard import scan_output, scan_prompt
from llm_guard.input_scanners import Anonymize, PromptInjection, Toxicity
from llm_guard.vault import Vault

client = OpenAI(api_key="OPENAIKEY")
vault = Vault()
input_scanners = [Anonymize(vault), Toxicity(), PromptInjection()]

prompt = "Make an SQL insert statement to add a new user to our database. Name is Jo
Email is test@test.com Phone number is 555-123-4567 and the IP address is 192.168.1
And credit card number is 4567-8901-2345-6789."

sanitized_prompt, results_valid, results_score = scan_prompt(input_scanners, prompt

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": sanitized_prompt},
    ],
)

# Sanitized Prompt:
# Make an SQL insert statement to add a new user to our database.
# Name is [REDACTED_PERSON_1]. Email is [REDACTED_EMAIL_ADDRESS_1] Phone number is
# [REDACTED_PHONE_NUMBER_1] and the IP address is [REDACTED_IP_ADDRESS_1].
# And credit card number is [REDACTED_CREDIT_CARD_RE_1].
```

6.1.4. Evaluate LLMs with `uptrain`

Evaluating LLMs can be tricky.

Luckily, `uptrain` offers a neat library to do that.

`uptrain` is a Python library to evaluate LLMs with 20+ preconfigured checks.

This includes the quality of the responses (completeness, validity,…), language proficiency (tonality, conciseness, …) and code hallucination.

It supports the biggest providers like OpenAI, Mistral, Claude and Ollama.

Skip to main content

```
!pip install uptrain
```

```
from uptrain import EvalLLM, Evals
import json

OPENAI_API_KEY = "*******"

data = [{
    'question': 'Which is the most popular global sport?',
    'context': "The popularity of sports can be measured in various ways, including
    'response': 'Football is the most popular sport with around 4 billion followers
}]

eval_llm = EvalLLM(openai_api_key=OPENAI_API_KEY)

results = eval_llm.evaluate(
    data=data,
    checks=[Evals.CONTEXT_RELEVANCE, Evals.FACTUAL_ACCURACY, Evals.RESPONSE_COMPLETI
)

print(json.dumps(results, indent=3))
```

6.1.5. Embed Any Type of File

These days, everything is about Embeddings and LLMs.

The Python library `embed-anything` makes it easy to generate embeddings from multiple sources like image, video, or audio.

It's built in Rust so it executes fast.

```
!pip install embed-anything
```

```
import embed_anything

data = embed_anything.embed_file("filename.pdf", embeder= "Bert")
embeddings = np.array([data.embedding for data in data])

data = embed_anything.embed_directory("test_files", embeder= "Clip")
embeddings = np.array([data.embedding for data in data])
```

Skip to main content

6.1.6. Structured LLM Output with `outlines`

Are you annoyed of unstructured outputs of LLMs?

Try `outlines`.

`outlines` is a Python library for structured generation of your LLMs.

There are multiple ways to enforce the output of the model like choices, type constraints, JSON output, and more.

```
!pip install outlines
```

```
import outlines

model = outlines.models.transformers("mistralai/Mistral-7B-Instruct-v0.2")

prompt = """You are a sentiment-labelling assistant.
Is the following review positive or negative?

Review: This restaurant is just awesome!
"""

generator = outlines.generate.choice(model, ["Positive", "Negative"])
answer = generator(prompt)
```

6.1.7. Evaluating RAG Pipelines with `Ragas`

How do you evaluate your RAG application?

Sure, you can look manually over your responses and see if it's what you want.

But, it's not scalable.

Instead, use `Ragas` in Python.

`Ragas` is a library providing evaluation techniques and metrics for your RAG pipeline like Context Precision/Recall, Faithfulness and answer relevancy

See below how easy it is to run `Ragas`.

```
!pip install ragas
```

```
from datasets import Dataset
import os
from ragas import evaluate
from ragas.metrics import faithfulness, answer_correctness

os.environ["OPENAI_API_KEY"] = "your-openai-key"

data_samples = {
    'question': ['When was the first super bowl?', 'Who won the most super bowls?'],
    'answer': ['The first superbowl was held on Jan 15, 1967', 'The most super bowls
    'contexts' : [['The First AFL–NFL World Championship Game was an American footba
    ['The Green Bay Packers...Green Bay, Wisconsin.','The Packers compete...Football
    'ground_truth': ['The first superbowl was held on January 15, 1967', 'The New Er
}

dataset = Dataset.from_dict(data_samples)

score = evaluate(dataset,metrics=[faithfulness,answer_correctness])
score.to_pandas()
```

6.1.8. Unified Reranker API with `rerankers`

An essential element of your RAG systems is reranking.

Reranking involves a reranking model that outputs a similarity score for each retrieved document and the user query.

The `rerankers` library gives you a unified API to use with popular vendors and models such as Cohere, Jina or T5.

The perfect API to easily test and replace many methods.

```
!pip install rerankers
```

```
from rerankers import Reranker

ranker = Reranker("t5")

results = ranker.rank(query="I love you", docs=["I hate you", "I really like you"],
```

6.1.9. Create Embeddings on your CPU with `fastembed`

My favourite library for creating embeddings:

`fastembed`, developed by Qdrant.

`fastembed` is a lightweight and fast library for using popular embedding models.

Without using your GPU.

It also integrates seamlessly with Qdrant's vector database.

I would like to see more supported models though, as `fastembed` has so much potential.

```
!pip install fastembed
```

```
from fastembed import TextEmbedding

documents = [
    "This is some",
    "example document",
]

embedding_model = TextEmbedding(model_name="jinaai/jina-embeddings-v2-small-en")

embeddings = list(embedding_model.embed(documents))
```

6.1.10. Convert Files to Markdown & JSON with `docling`

Preparing your data for LLMs is a crucial step in RAG applications.

`docling` simplifies this step for you by converting popular document formats like PDF or PPT to

It uses two models, layout analyis model and table structure recognition model, to process the files.

```
!pip install docling
```

```
from docling.document_converter import DocumentConverter

source = "https://arxiv.org/pdf/2408.09869"
converter = DocumentConverter()
result = converter.convert(source)
print(result.document.export_to_markdown())
# Output: "## Docling Technical Report[...]"
```

6.1.11. Simple Chunking Library with `chonkie`

Having a great chunking library without installing 500 MB of subdependencies is my childhood's dream.

Luckily, `chonkie` provides you with the most important chunking strategies.

Currently, it supports:

- Token chunker
- Word chunker
- Sentence chunker
- Semantic chunker
- Semantic Double-Pass Merge chunker

```
!pip install chonkie
```

```
from chonkie import SemanticChunker

chunker = SemanticChunker(
    embedding_model="all-minilm-l6-v2",
    chunk_size=512,
    similarity_threshold=0.7
)

chunks = chunker.chunk("Some text with semantic meaning to chunk appropriately.")
for chunk in chunks:
    print(f"Chunk: {chunk.text}")
    print(f"Number of semantic sentences: {len(chunk.sentences)}")
```

6.1.12. Type-Safe Agentic AI with `pydantic-ai`

A type-safe Python agent framework was on my Christmas wishlist.

Luckily, **PydanticAI** came out earlier.

PydanticAI bridges the gap between LLMs and structured data validation.

Type-safety is my favourite feature, while making structured responses and using custom models easy.

Definitely something I have to go much deeper into it.

```
!pip install pydantic-ai
```

```
from pydantic_ai import Agent

agent = Agent(
    'gemini-1.5-flash',
    system_prompt='Be concise, reply with one sentence.',
)

result = agent.run_sync('Where does "hello world" come from?')
print(result.data)
"""
The first known use of "hello, world" was in a 1974 textbook about the C programming
"""
```

Skip to main content

6.1.13. Convert Files into Markdown with `markitdown`

Preparing your data for LLMs can be hard.

Different file formats like PPT, Excel or Audio files need different preprocessing steps.

Luckily, with **markitdown** from Microsoft, this is easy.

markitdown converts various file formats to Markdown:

Currently, it supports:

- PDF
- PPT
- Word
- Excel
- Images
- Audio
- HTML
- JSON, XML
- ZIP files

What I really like is that for image descriptions, you can use LLMs and even adjust the prompt for it.

Thanks to Jimi Vaubien for showing me this gem.

```
!pip install markitdown
```

```
from markitdown import MarkItDown
from openai import OpenAI

client = OpenAI()
md = MarkItDown(llm_client=client, llm_model="gpt-4o", llm_prompt="Describe the ima
result = md.convert("example.jpg")
print(result.text_content)
```

Skip to main content

6.1.14. Route Queries Intelligently with `RouteLLM`

Using LLMs like o1/o3 or Claude Sonnet for every task is a waste of money.

For simple tasks, you may switch to a much simpler and lightweight model to save on costs and compute time.

With RouteLLM, queries will be routed intelligently to the right LLM for the job.

RouteLLM acts as a traffic controller for your LLM workflows.

Instead of blindly sending all requests to the most expensive model, it dynamically routes queries based on: -> Complexity: Simple tasks for smaller, cheaper models -> Accuracy needs: Critical tasks for heavyweight models

And it’s open-source!

See below for a simple example: -> We use GPT-4 as our strong model and Mixtral 8x7B as our weak model -> The router model **router-mf-0.11593** is the default routing model and sufficient for basic query routing. The threshold **0.11593** can be calibrated by yourself if you want to to say that e.g. ~30 % of the queries will be routed to the strong model.

```
!pip install "routellm[serve,eval]"
```

```
import os
from routellm.controller import Controller

os.environ["OPENAI_API_KEY"] = "sk-XXXXXX"
os.environ["ANYSCALE_API_KEY"] = "esecret_XXXXXX"

client = Controller(
  routers=["mf"],
  strong_model="gpt-4-1106-preview",
  weak_model="anyscale/mistralai/Mixtral-8x7B-Instruct-v0.1",
)

response = client.chat.completions.create(
  model="router-mf-0.11593",
  messages=[
    {"role": "user", "content": "Hello!"}
  ]
)
```

7. NumPy Tips and Tricks

7.1. NumPy Tips and Tricks

7.1.1. Achieve Reproducibility with `np.random.RandomState()`

Reproducibility in Data Science projects is key.

For larger projects, use `numpy.random.RandomState()` to construct a random number generator.

Using `numpy.random.seed()` sets the global random seed, which affects all uses to the `numpy.random.*` module.

Imported packages or other modules can reset the global random seed to another one.

This can result in undesirable and unreproducible results across your project.

With `numpy.random.RandomState()`, you are not relying on the global random state anymore (which could be resetted).

```
import numpy as np

rng = np.random.RandomState(1234)

print(rng.rand(3))
```

8. Pandas Tricks and Tips

8.1. Pandas Tips and Tricks

8.1.1. Filter Pandas in a readable format

Do you need a more readable way to filter Dataframes in Pandas?

Try `df.query()`.

You can specify the condition using a string.

This can be sometimes more convenient and readable than boolean indexing.

And it's fast, due to the optimized Cython-based code used under the hood.

```
import pandas as pd

df = pd.DataFrame({"A": [1,2,3], "B": [4,5,6]})

filtered_df = df.query("A > 1 & B < 6")
```

8.1.2. Get frequency of time series with `pd.infer_freq`

When working with time series, it is useful to know the frequency of the data.

But especially in larger datasets, it might be difficult to tell the frequency immediately.

To get the frequency of a time series in Pandas, use `pd.infer_freq()`.

Skip to main content

It infers the most likely frequency given the input index.

Below you can see how to infer the most likely frequency for a `DatetimeIndex`.

```
import pandas as pd
time_index = pd.date_range(start='1/1/2020 20:00:00', end='10/1/2020 00:00:00')

pd.infer_freq(time_index)
'D'
```

8.1.3. Change the Plotting Backend

By default, Pandas uses matplotlib as its plotting backend.

You can change it to, let's say Plotly, with one line of code.

See below how to do that with only one line.

```
import pandas as pd

pd.options.plotting.backend = 'plotly'

df = pd.DataFrame(dict(a=[5,7,9,3], b=[1,6,4,10]))
fig = df.plot()
fig.show()
```

8.1.4. Style your DataFrames

Did you know you can style your DataFrames in Pandas?

You just have to define a condition to apply colors in a function.

And use `DataFrame.style.applymap()` to apply the condition.

```
import pandas as pd

sales_data = sales_data = {
    'Product Name': ['Product A', 'Product B', 'Product C', 'Product D'],
    'Revenue': [10000, 5000, 15000, 1000],
}

sales_df = pd.DataFrame(sales_data)

# Apply styles to the dataframe
def coloring(val):
    color = 'red' if val <= 10000 else 'green'
    return 'background-color: %s' % color

sales_df.style.applymap(coloring, subset=['Revenue'])
```

8.1.5. Set Precision of Displayed Floats

In Pandas, you can control the precision of the displayed values.

Just use the .set_option() function.

```
import pandas as pd

pd.set_option('display.precision', 2)

data = {'Value': [1.2343129, 5.8956701, 6.224289]}
df = pd.DataFrame(data)
```

8.1.6. Faster I/O with Parquet

Whenever you work with bigger datasets, please avoid using CSV format (or similar).

CSV files are text files, which are human-readable, and therefore a popular option to store data.

For small datasets, this is not a big issue.

But, what if your data has millions of rows?

It can get really slow to do read/write operations on them.

Skip to main content

They consist of 0s and 1s and are not meant to be human-readable but to be used by programs that know how to interpret them.

Because of that, binary files are more compact and consume less space.

Parquet is one popular binary file format, which is more memory-efficient than CSVs.

```
import pandas as pd

# Shape: (100000000, 5)
df = pd.DataFrame(...)

# Time: 1m 58s
df.to_csv("data.csv")

# Time: 8s
df.to_parquet("data.parquet")
```

8.2. Utility Libraries for Pandas

8.2.1. Speed up Pandas' `apply()`

Don't use `.apply()` in Pandas blindly!

`.apply()` is used to apply operations on all the elements in a dataframe (row-wise or column-wise).

It's the most obvious choice, but there is a better option:

Instead, use the **Swifter** package.

Swifter tries to pick up the best way to implement the `.apply()` function by either:

- Vectorizing your function
- Parallelizing using Dask
- Using `.apply()` from Pandas if the dataset is small.

That gives your function a huge boost.

Skip to main content

```
!pip install swifter
```

```
import swifter
import pandas as pd

df = pd.DataFrame(...)

def my_function(input_value):
        ...
    return output_value

df["Column"] = df["Column"].swifter.apply(lambda x: my_function(x))
```

8.2.2. Reduce DataFrame Memory with `dtype_diet`

By default, Pandas DataFrames don't use the smallest data types for its columns.

This results in unnecessary memory usage.

Changing data types can drastically reduce the memory usage of your DataFrame.

Using `dtype_diet`, you can automatically change the data types to the smallest (and most memory-efficient) one.

```
!pip install dtype-diet
```

```
from dtype_diet import optimize_dtypes, report_on_dataframe
import pandas as pd

df = pd.read_csv("")
# Get Recommendations
proposed_df = report_on_dataframe(df, unit="MB")
new_df = optimize_dtypes(df, proposed_df)
print(f'Original df memory: {df.memory_usage(deep=True).sum()/1024/1024} MB')
print(f'Propsed df memory: {new_df.memory_usage(deep=True).sum()/1024/1024} MB')
```

8.2.3. Validate Pandas DataFrames with `pandera`

Skip to main content

Try `pandera`.

`pandera` is a data validation library for Pandas DataFrames and Series.

It provides a convenient way to define and enforce data quality constraints.

You can even define complex constraints or use the in-built constraints.

```
!pip install pandera
```

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema({
    "name": pa.Column(pa.String),
    "age": pa.Column(pa.Int, checks=[
        pa.Check(lambda x: x > 0, element_wise=True),
        pa.Check(lambda x: x < 100, element_wise=True)
    ]),
})

data = {
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 40, 200],
}
df = pd.DataFrame(data)


schema.validate(df)
```

8.2.4. Boost Pandas' Performance With One Line With `modin`

If you already have a large codebase based on Pandas, think again.

You can also use `modin` as a drop-in replacement for Pandas, with a 3X-5X speed-up.

Just install `modin` and replace the import statement.

It's maybe not as fast as polars, but you will save hours of development time and gain some performance boost.

Skip to main content

```
!pip install "modin[all]"
```

```
import modin.pandas as pd

df = pd.read_csv("")
```

8.2.5. Chat with your Dataframe with `PandasAI`

You can chat with your Pandas dataframe with a few lines of code.

With `PandasAI`, you can use LLMs to analyze your data, generate visuals, and create a report with your words.

Currently, `PandasAI` supports popular LLMs from providers like OpenAI, Anthropic, Google, Amazon, or Ollama for local LLMs.

```
!pip install pandasai
```

```
from pandasai import SmartDataframe
from pandasai.llm import OpenAI
from pandasai.helpers.openai_info import get_openai_callback

llm = OpenAI()

df = SmartDataframe("data.csv", config={"llm": llm, "conversational": False})

with get_openai_callback() as cb:
    response = df.chat("Calculate the sum of the gdp of north american countries")
```

9. Polars

9.1. Polars Tips & Tricks

9.1.1. Plugin for Data Science Functions

Polars gains increasing popularity.

If you already ditched pandas for it, you don't have to rewrite all of your functions in Polars again.

`polars-ds`, a community plugin, has reimplemented common functions for Data Scientists like:

- Statistical tests (t-test, …),
- String similarities (Levenshtein, …)
- Loss Functions and metrics (ROC, R2, L1, Huber, …)

```
!pip install polars_ds
```

```
import polars_ds
import polars as pl

df = pl.DataFrame(...)

# Calculate Loss and Metrics
df.group_by("dummy_groups").agg(
    pl.col("actual").num_ext.l2_loss(pl.col("predicted")).alias("l2"),
    pl.col("actual").num_ext.bce(pl.col("predicted")).alias("log loss"),
    pl.col("actual").num_ext.binary_metrics_combo(pl.col("predicted")).alias("combo"
).unnest("combo")
```

9.1.2. Plugin for Fitting Linear Models

In Polars, you can fit linear models with the `polars-ols` extension.

You can use ordinary, weighted or regularized least squares like Lasso or Elastic Net.

It can be 2x-88x times faster than popular libraries like sklearn or statsmodels.

Skip to main content

```
!pip install polars-ols
```

```
import polars as pl
import polars_ols as pls

lasso_expr = pl.col("y").least_squares.lasso("x1", "x2", alpha=0.0001, add_intercept

predictions = df.with_columns(lasso_expr.round(2).alias("predictions_lasso"))
```

10. Python Tips and Tricks

10.1. Pure Python + Built-in libraries

10.1.1. Make your numbers more readable

Do you want to make your numbers in Python more readable?

Use underscores such as `1_000_000`.

Underscores in numbers in Python are used to make large numbers more readable.

The underscores are ignored by the interpreter and do not affect the value of the number.

See below for a small example.

Note: Using two consecutive underscores is not allowed!

```
big_number = 1_000_000_000_000

print(big_number)
```

10.1.2. Get Query Parameters with `urllib.parse`

How to extract query parameters from an URL in Python?

Skip to main content

Query parameters are the extra pieces of information you can add to a URL to change how a website behaves.

They come after a ‘q’ character in the URL and are made up of key-value pairs.

The key describes what the information is for and the value is the actual information you want to send.

Query parameters are extra pieces of information that you can add to a URL to change how a website behaves.

To extract those query parameters in Python, use `urllib`.

`urllib` provides functions to extract the query and its parameters for you.

```
from urllib import parse
url = 'https://play.google.com/store/apps/details?id=com.happy.mood.diary&hl=de&gl=U

# Outputs "id=com.happy.mood.diary&hl=de&gl=US"
query = parse.urlparse(url).query

# Outputs "{'id': ['com.happy.mood.diary'], 'hl': ['de'], 'gl': ['US']}"
parameters = parse.parse_qs(query)
```

10.1.3. Zip iterables to the longest iterable

Don’t use `zip()` in Python.

When you have, let’s say, two lists of unequal length, zip() will return an iterable with as many elements as the shortest list.

Instead, use `itertools.zip_longest()`.

It will “pad” any shorter lists (or other iterables) with a fill value so that the returned iterable has the same length as the longest iterable.

You will not lose any elements!

```
from itertools import zip_longest

a = [1,2,3,4]
b = [5,6,7]

# zip(): One element is missing
for aa, bb in zip(a, b):
  print(aa, bb)
'''
1 5
2 6
3 7
'''

# zip_longest()
for aa, bb in zip_longest(a, b):
  print(aa, bb)

'''
1 5
2 6
3 7
4 None
'''
```

10.1.4. Improve readability with Named slices

Do you want to make your code more readable in Python?

Use **named slices**.

They are reusable and make your code less messy.

Especially when there is a lot of slicing involved.

```
LETTERS = slice(0,2)
NUMS = slice(2,6)
CITY = slice(6, None)

code_1 = "LH1234 BLN"
code_2 = "LH7672 MUC"

print(code_1[LETTERS], code_1[NUMS], code_1[CITY])
print(code_2[LETTERS], code_2[NUMS], code_2[CITY])
```

Skip to main content

10.1.5. Pythonic way for matrix multiplication

Did you know the '@' operator performs **matrix multiplication** in Python?

Normally, you would use `numpy.matmul()`.

But since Python 3.5, you can also use the '@' operator as a more readable way.

```
import numpy as np

a =  np.array([[1, 2],
               [4, 1],
               [3, 4]])
b = np.array([[4, 5],
               [1, 0]])

a @ b

np.matmul(a, b)
```

10.1.6. Use Guard Clauses for better If statements

Stop nesting your If-statements.

Instead, use the **Guard Clause** technique.

The **Guard Clause** pattern says you should terminate a block of code early by checking for invalid inputs or edge cases at the beginning of your functions.

Below you can see how we can make our If-statements more readable by checking for the conditions and immediately return None if it's false.

```python
# Old way
def calculate_price(quantity, price_per_unit):
    if quantity > 0:
        if price_per_unit > 0:
            return quantity * price_per_unit
        else:
            return None
    else:
        return None

# Better way with Guard Clause
def calculate_price(quantity, price_per_unit):
    if quantity <= 0:
        return None
    if price_per_unit <= 0:
        return None
    return quantity * price_per_unit
```

10.1.7. Hide Password Input from User

Do your Python Command-line apps include collecting secret keys or passwords?

Use `getpass`, from the Python Standard Library.

`getpass` ensures the user's inputs will not be echoed back to the screen.

A cool tool for your next Command-line app!

```python
import getpass

username = getpass.getuser()

print(f"{username} is logged in.")

password = getpass.getpass()

print(f"Password entered is: {password}")
```

10.1.8. Turn Classes into Callables

Did you know you can have callable objects in Python?

Skip to main content

```
class Example:
  def __call__(self):
    print("I was called!")

example_instance = Example()
example_instance() # Will call the __call__ method
```

10.1.9. Wrap Text with `textwrap`

Do you want to format your text output easily?

Use `textwrap`!

`textwrap` lets you wrap paragraphs, adjust line widths and handle indentation.

```
import textwrap

text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec tellus

# Wrap text to a specific line width
wrapped_text = textwrap.wrap(text, width=30)

for line in wrapped_text:
    print(line)
```

10.1.10. Add LRU Caching to your Functions

Python offers a neat way to add caching to functions.

You just need to add the `lru_cache` decorator from `functools`.

It takes a `maxsize` argument that specifies the maximum number of results to cache.

When the function is called with the same arguments, it first checks the cache and returns the cached result if available.

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

fibonacci(10)
```

10.1.11. Cache Methods in Classes with `functools.cached_property`

To cache the result of a method of a class, use `functools.cached_property`.

It transforms a method into a property where its value is computed only once and then cached.

```python
from functools import cached_property

class MyClass:
    @cached_property
    def expensive_operation(self):
        # Perform some expensive computation here
        return result

my_object = MyClass()

print(my_object.expensive_operation)
```

10.1.12. For-Else Loops in Python

You probably know about If-Else.

But do you know about `For-Else`?

You can add an else-block which will be executed only if the loop completes all its iterations without executing a break statement.

See the small example below, where the else-block will be executed after nothing was found in the for-loop

Skip to main content

```
fruits = ['apple', 'orange', 'pear']

for fruit in fruits:
    if fruit == 'banana':
        print("Found the banana!")
        break
else:
    print("Banana not found in the list.")
```

10.1.13. Heaps in Python

Want an efficient way to retrieve the N smallest/largest elements in Python?

Use `heapq`.

`heapq` is a standard library for working with heaps.

A heap is a specialized tree-based data structure that satisfies the heap property.

In a heap, the value of each node is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the values of its children.

Therefore, retrieving the N smallest/largest elements can be done in a fast way.

```
import heapq

data = [5, 9, 2, 1, 6, 3, 8, 4, 7]

largest = heapq.nlargest(3, data)
print("Largest elements:", largest)  # Output: [9, 8, 7]

smallest = heapq.nsmallest(3, data)
print("Smallest elements:", smallest)  # Output: [1, 2, 3]
```

10.1.14. Difference between `__str__` and `__repr__`

Do you know the difference between `__str__` and `__repr__`?

In Python, both methods are used to define string representations of an object.

Skip to main content

The `__str__` method is intended to provide a human-readable string representation of the object.

While the `__repr__` method is intended to provide a detailed string representation.

See the example below with the built-in datetime library.

We see that the `__repr__` method of the datetime class has a detailed and unambiguous output.

```
import datetime
today = datetime.datetime.now()

print("str:", str(today))

print("repr: ", repr(today))

# str:   2023-07-02 21:21:00.771969
#repr:  datetime.datetime(2023, 7, 2, 21, 21, 0, 771969)
```

10.1.15. Neat Way to Merge Dictionaries

Do you need a clean way to merge two dictionaries in Python?

Use the `|` operator (available since Python 3.9)

Unlike other methods (like the .update method or unpacking), it offers a neat way to merge dictionaries.

See below for a small example.

```
dict_1 = {"A": 1, "B": 2}
dict_2 = {"C": 3, "D": 4}

dict_1 | dict_2
```

10.1.16. Switch Case Statements in Python

Did you know Python has switch-cases too?

They're called match-case and it was introduced in Python 3.10+

It allows you to perform (complex) pattern matching on values and execute code blocks based on the matched patterns.

```
def match_example(value):
    match value:
        case 1:
            print("Value is 1")

        case 2 | 3:
            print("Value is 2 or 3")

        case 4:
            print("Value is 4")

        case _:
            print("Value is something else")

match_example(2)
```

10.1.17. Walrus Operator in Python

Do you know about the Walrus operator in Python?

The Walrus operator allows you to assign a value to a variable as part of an expression.

```
# With Walrus Operator
if (name := input("Enter your name: ")):
    print(f"Hello, {name}!")
else:
    print("Hello, anonymous!")
```

```
# Without Walrus Operator
name = input("Enter your name: ")
if name:
    print(f"Hello, {name}!")
else:
    print("Hello, anonymous!")
```

10.1.18. Count Occurrences in an Iterable with `Counter`

Instead, use `collections.Counter` in Python.

`Counter` is used for counting the occurrences of elements in an iterable.

```
# With Counter
from collections import Counter

fruits = ["apple", "banana", "apple", "orange", "banana", "kiwi"]

fruit_counter = Counter(fruits)

# Output: Counter({'apple': 2, 'banana': 2, 'orange': 1, 'kiwi': 1})
```

```
# Without Counter
fruits = ["apple", "banana", "apple", "orange", "banana", "kiwi"]

fruit_counts = {}

for fruit in fruits:
    if fruit in fruit_counts:
        fruit_counts[fruit] += 1
    else:
        fruit_counts[fruit] = 1
```

10.1.19. Set Default Values for Dictionaries with `defaultdict`

Don’t use dictionaries in Python.

Instead use `defaultdicts`.

`defaultdict` is similar to dictionaries, but it allows you to set a default value for new keys that haven’t been added yet.

This is useful when you want to perform operations without checking if a key exists.

```
from collections import defaultdict

d = defaultdict(list)

d["Fruits"].append("Kiwi")
d["Cars"].append("Mercedes")

print(d["Fruits"])
print(d["Cars"])
print(d["Animals"])

'''
Output:
['Kiwi']
['Mercedes']
[]
'''
```

10.1.20. More Structured Tuples with `namedtuple`

You don't know about `namedtuple`?

`namedtuple` are similar to regular tuples, but have named fields.

This makes them more self-documenting and easier to work with.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(x=3.2, y=1.0)
print(p.x, p.y)
```

10.1.21. Mutable Default Values for Function Arguments

One big mistake in Python:

Using mutable default values for function arguments.

When using mutable objects like a list as a default value, you have to be careful.

See the example below, where the default list is shared among all calls to the function.

To fix this, set the default value to None and create a new list inside the function if no list is provided.

```
# Dangerous behaviour:
def increment_numbers(numbers=[]):
    for i in range(3):
        numbers.append(i)
    print(f"Numbers: {numbers}")

increment_numbers()  # Output: Numbers: [0, 1, 2]
increment_numbers()  # Output: Numbers: [0, 1, 2, 0, 1, 2]


# Better:
def increment_numbers(numbers=None):
    if numbers is None:
        numbers = []
    for i in range(3):
        numbers.append(i)
    print(f"Numbers: {numbers}")

increment_numbers()  # Output: Numbers: [0, 1, 2]
increment_numbers()  # Output: Numbers: [0, 1, 2]
```

10.1.22. Optimize Your Python Objects with `__slots__`

Optimize your Python classes with this trick.

When creating objects, a dynamic dictionary is created too to store instance attributes.

But, what if you know all of your attributes beforehand?

With **`__slots__`**, you can specify which attributes your object instances will have.

This saves space in memory and prevents users from creating new attributes at runtime.

See below how setting a new instance attribute will throw an error.

```
class Point:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

point = Point(2,4)
point.z = 5 # Throws an error
```

10.1.23. Modify Print Statements

When printing multiple elements in Python, it starts a new line by default.

But, you can change this default to whatever you like by setting the `end` parameter in `print()`.

See below where we change the `end` parameter to space.

```
sports = ["football", "basketball", "volleyball", "tennis", "handball"]

for sport in sports:
    print(sport, end=" ")
```

10.1.24. Type Variables in Python 3.12

One cool feature in Python 3.12:

The support for Type Variables.

You can use them to parametrize generic classes and functions.

See below for a small example where our generic class is parametrized by T which we indicate with [T].

```
class Stack[T]:
    def __init__(self) -> None:
        self.items: List[T] = []

    def push(self, item: T) -> None:
```

Skip to main content

10.1.25. Enumerations in Python

To make your Python Code cleaner, use `Enums`.

`Enums` (or enumerations) are a way to represent a set of named values as symbolic names.

It provides a way to work with meaningful names instead of raw integers or weird string constants.

Python supports enums with its standard library.

```
# Option 1
from enum import Enum, auto

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(Color.RED.value)  # Output: 1

# Option 2: auto() assigns unique values starting by 1
class Status(Enum):
    PENDING = auto()
    APPROVED = auto()
    REJECTED = auto()

print(Status.PENDING.value) # Output: 1
```

10.1.26. Never use `eval`

One big mistake in Python:

Using `eval` in your code.

`eval`, like the name suggests, evaluates a Python expression.

The big problem: One could inject malicious code which will be evaluated.

Skip to main content

```
def calculate(expression):
    return eval(expression)

print(calculate("5 + 5"))

# DON'T RUN THIS LINE OF CODE
print(calculate("__import__('os').system('rm -rf /')"))
```

10.1.27. Never use `import *`

One nooby Python mistake:

Running `import *`.

You will only pollute your namespace, leading to potential naming conflicts.

Your code also becomes less readable and maintainable when it's not clear where each name is coming from.

Remember "*Explicit is better than implicit*".

```
# Ugly
from module import *

# Better
from module import Class1, Class2, Class3
```

10.1.28. Control What Gets Imported

How to control what gets imported when you use `from module import *` in Python?

`__all__` lets you specify which symbols should be imported when you run `from module import *`.

By defining `__all__` in your module, you can explicitly control what gets exposed.

Skip to main content

```python
# mymodule.py
__all__ = ['my_function', 'MyClass']

def my_function():
    pass

class MyClass:
    pass

def internal_helper():
    pass

# main.py
from mymodule import *

my_function()
internal_helper() # Error
```

10.1.29. Make Fields Keyword-Only in `dataclasses`

If you want to have keyword-only fields in Python, use the `kw_only` option from `dataclasses`.

This will throw an error whenever you set parameters without the keyword.

```python
from dataclasses import dataclass
@dataclass(kw_only=True)
class Example:
    a: int
    b: int

example = Example(a=1, b=2)
example = Example(1,2) # Error
```

10.1.30. Keyword-only and Positional-only Arguments

How do you define keyword-only and positional arguments in Python?

- **Keyword-only** arguments: Specified after an **asterisk** in the function definition. Perfect for readability and when passing optional arguments to a function.
- **Positional only**: Specified before a **slash** in the function definition. Perfect for functions where argument names may change

Skip to main content

```python
# Keyword-only
def greet(*, name, greeting="Hello"):
    print(f"{greeting}, {name}!")

# This works:
greet(name="Alice")

# This raises an error:
greet("Alice")

# Positional-only
def greet(name, greeting="Hello", /):
    print(f"{greeting}, {name}!")

# This works:
greet("Alice")

# This raises an error:
greet(name="Alice")
```

10.2. Utilities for Python

10.2.1. Speed up For-Loop with `joblib.Parallel`

Don’t use plain for loops in Python.

`Joblib` provides a `Parallel` class to write parallel for loops using multiprocessing.

Below you can see an example of how to use it with all of your processors to speed up your calculations.

```python
from joblib import Parallel, delayed

def process_image(path):
   ...
    return image

 image_paths = ["path1.jpg", "path2.jpg"]

result = Parallel(n_jobs = -1, backend = "multiprocessing")(
  delayed(process_image)(path) for path in image_paths
)
```

10.2.2. Work with datetimes easily with `pendulum`

Tired of the difficulties of working with dates and times in Python?

Try `pendulum`!

`pendulum` takes the built-in `datetime` library from Python to the next level with its intuitive and human-friendly way of handling dates and times.

This includes easy timezone manipulation, daylight saving time calculations, and more!

```
#!pip install pendulum
import pendulum

dt = pendulum.now()
print(dt.to_iso8601_string())
# Output: 2023-02-08T13:44:23.798316+01:00

now_in_london = pendulum.now('Europe/London')
print(now_in_london)
# Output: 2023-02-08T12:44:23.799317+00:00

past = pendulum.now().subtract(minutes=8)
print(past.diff_for_humans())
# Output: 8 minutes ago

delta = past - pendulum.now().subtract(weeks=1)
print(delta.in_words())
# Output: 6 days 23 hours 51 minutes 59 seconds
```

10.2.3. Prettify your Data Structures with `pprint`

Using `print()` on your data structures can give you ugly outputs.

With `pprint`, you can print data structures in a pretty way

Don't use plain `print()` for printing data structures anymore.

```
from pprint import pprint

data = [ (i, { 'a':'A',
               'b':'B',
               'c':'C',
               'd':'D',
               'e':'E',
               'f':'F',
               'g':'G',
               'h':'H',
               })
         for i in range(2)
         ]

pprint(data)
```

10.2.4. Easy Logging with `loguru`

Are you too lazy for logging?

`loguru` makes logging in Python easy.

It comes with pre-built formats and colors so you don't have to set them manually.

A more elegant alternative to Python's standard logging module.

```
from loguru import logger

def main():
    logger.debug("DEBUG message")
    logger.info("INFO message")
    logger.warning("WARNING message")
    logger.error("ERROR message")
    logger.critical("CRITICAL message")


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

10.2.5. Generate Truly Random Numbers

How to generate `truly` random numbers?

Skip to main content

One problem with random number generators in your favourite programming language is:

They are `pseudo-random`.

That means they are generated using a deterministic algorithm.

If you want to generate truly random numbers,

You have to make an API request to `random(dot)org`.

They create random numbers based on atmospheric noise.

See below how you can use it with Python.

What do the Query Parameters mean?

- `num=1`: Specifies that only one integer should be generated.
- `min=1`: Specifies that the minimum value for the generated integer should be 1.
- `max=1000`: Specifies that the maximum value for the generated integer should be 1000.
- `col=1`: Specifies that the output should be in a single column. Only useful for displaying purposes when you are generating more than one number.
- `base=10`: Specifies that the generated integers should be in base 10 (decimal).
- `format=plain`: Specifies that the output should be in plain text format.
- `rnd=new`: Specifies that a new random sequence should be used for each request.

```
import requests

url = "https://www.random.org/integers/?num=1&min=1&max=1000&col=1&base=10&format=pl

response = requests.get(url)
print(response.text)
```

10.2.6. Powerful Dictionaries with `python-benedict`

`python-benedict` is a library for dictionaries on steroids.

You can access values of your nested dictionary with a neat syntax, perform searching and GroupBy,

Skip to main content

Of course, Pandas offers similar functionalities, but takes also much memory when installing. But **python-benedict** is more suitable for more unstructured data.

```
!pip install python-benedict
```

```
from benedict import benedict

my_dict = benedict({
    'person': {
        'name': 'John',
        'age': 25,
        'address': {
            'street': '123 Main St',
            'city': 'New York',
            'country': 'USA'
        }
    }
})

my_dict.get("person.address.street")

my_dict.flatten()
```

10.2.7. Clear Exception Output with `pretty_errors`

Are you annoyed by the unclear Python error messages?

Try `pretty_errors`.

It's a library to prettify Python exception output to make it more readable and clear.

It also allows you to configure the output like changing colors, separator character, displaying locals, etc..

You just have to:

Install it: `pip install pretty_errors` Import it: `import pretty_errors`.

```
!pip install pretty_errors
```

Skip to main content

```
import pretty_errors

num = 1 / 0
```

10.2.8. Deprecate Functions with `deprecated`

How to deprecate functions in Python?

You can use the library `deprecation` which offers decorators to wrap functions.

Proper warnings in documentation and in the console are displayed.

```
!pip install deprecation
```

```
from deprecation import deprecated

@deprecated(deprecated_in="0.10", removed_in="1.0", current_version=__version__, de
def my_func():
  print("Hello World")

my_func()
# UnsupportedWarning: my_func is unsupported as of 1.0. Use my_func2 instead
```

10.2.9. Case Insensitive Dictionaries

Are you annoyed by case-sensitive dictionaries in Python?

Try **CaseInsensitiveDicts**.

As the name says, you can access values by the key without paying attention to lowercase or uppercase.

It's useful for everyone who is scraping API data.

```
from requests.structures import CaseInsensitiveDict

data: dict[str, str | int] = CaseInsensitiveDict(
    {
        "accept": "application/json",
        "content-type": "application/json",
        "User-Agent": "Mozilla/5.0",
    }
)

print(data.get("Accept"))
```

10.3. Tooling for Python Projects

10.3.1. `uv`: Super-fast pip Alternative

Forget plain pip for installing packages.

Use `uv` for Python package installing.

`uv` is a blazingly fast package installer and resolver, written in Rust for high performance.

It is a drop-in replacement for pip and pip-tools, being up to 115x faster.

`uv` is still in an early phase, but it's interesting to see where it goes.

```
!curl -LsSf https://astral.sh/uv/install.sh | sh
```

```
!uv pip install requests
```

11. Scraping Tips and Tricks

11.1. Scraping Tips and Tricks

11.1.1. Manage your Webdrivers with `webdriver-manager`

When using `Selenium` with Python, you probably did the following:

- Download Chromedriver Binary
- Unzip it
- Set the path to the driver

This is annoying.

- The Path can be changed
- You have to somehow manage those browser drivers for each OS
- Check if new updates for drivers are released

Instead of doing this manually, use `webdriver-manager`.

It makes managing binaries for different browsers easy.

`webdriver-manager` downloads binaries automatically for you.

So you don't have to go through the pain of doing it manually.

To use it in your project, see the example below. It's straightforward and saves you time and energy.

Especially when you integrate `Selenium` in your CI/CD Pipeline.

By default, `webdriver-manager` installs the latest version.

But you can also define a specific version of the driver.

```
!pip install webdriver-manager
```

```
# Old way
from selenium import webdriver
driver = webdriver.Chrome('path/to/driver.exe')

# New way with webdriver-manager and Selenium 4
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

# New way with webdriver-manager and Selenium 3
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager

driver = webdriver.Chrome(ChromeDriverManager().install())

# Use specific version
driver = webdriver.Chrome(executable_path=ChromeDriverManager("<your_version>").ins
```

11.1.2. Speed up your Scraping with disabling image loading

Do you want to speed up your web scraper?

Disable image loading!

Disabling image loading while scraping is a great way to speed up your scraper.

You are wasting a lot of connection bandwidth.

To disable image loading in Selenium, you only have to set one option (like below).

This will save you time and money.

```
from selenium import webdriver

chrome_options = webdriver.ChromeOptions()

chrome_options.add_argument('--blink-settings=imagesEnabled=false')

driver = webdriver.Chrome(chrome_options=chrome_options)
driver.get("https://www.instagram.com/")
```

11.1.3. AI-Powered Web Scraper with `scrapegraph-ai`

Do you want to let AI scrape your website?

Use `scrapegraph-ai`.

This library uses LLM and direct graph logic to scrape websites by only providing the information you need.

See below where we give it a prompt and an URL.

It also supports multi-page scraper that extracts information from the top n search results of a search engine.

```
!pip install scrapegraphai
```

```
from scrapegraphai.graphs import SmartScraperGraph

graph_config = {
    "llm": {
        "api_key": OPENAI_API_KEY,
        "model": "gpt-3.5-turbo",
        "temperature":0,
    },
    "verbose":True,
}

smart_scraper_graph = SmartScraperGraph(
    prompt="List me all the projects with their descriptions.",
    source="https://perinim.github.io/projects/",
    config=graph_config
)

'''
Output
{
  "projects": [
    {
      "title": "Rotary Pendulum RL",
      "description": "Open Source project aimed at controlling ..."
    },
    {
      "title": "DQN Implementation from scratch",
      "description": "Developed a Deep Q-Network algorithm to train a ..."
    },
    {
      "title": "Multi Agents HAED",
      "description": "University project which focuses ...."
    },
    {
      "title": "Wireless ESC for Modular Drones",
      "description": "Modular drone architecture ..."
    }
  ]
}
'''
```

12. Terraform

12.1. Terraform

12.1.1. Static Code Analyzer with `TFLint`

As your Terraform codebase grows, it's more prone to errors.

And they can become nasty.

To analyze your files, use `TFLint`.

`TFLint` is a linting tool to check for common smells and issues before you even run your code.

You can even define custom rules to tailor it to your team's standards.

```
tflint

Error: "type_abc" is an invalid value as instance_type (aws_instance_invalid_type)

  on main.tf line 15:
   15: instance_type = "invalid_type"
When you run TFLint again, you'll receive an error message indicating that the
instance_type value is invalid. TFLint identifies this issue without applying the Te
helping you catch errors early in your development workflow.
```

12.1.2. Check for Security Risks with `Tfsec`

To check your Terraform files for common security issues, use `Tfsec`.

`TFsec` is a static code analyzer to detect security risks, with hundreds of built-in rules.

It's easy to set-up and run. Perfect for CI pipelines too.

```
brew install tfsec
tfsec .

Result #1 CRITICAL Storage account uses an insecure TLS version.
```

```
  deployment/modules/storage_account/main.tf:1-8
   via deployment/main.tf:23-28 (module.storage_account)
```

```
    1    resource "azurerm_storage_account" "storage_account" {
    2      name                     = var.account_name
    3      resource_group_name      = var.resource_group_name
    4      location                 = var.location
    5      account_kind             = "Storage"
    6      account_tier             = "Standard"
    7      account_replication_type = "LRS"
    8    }
```

12.1.3. Generate Least Privileges with `Pike`

When managing your infrastructure, follow the Principle of Least Privilege (PoLP).

PoLP says that a user should only have the minimum level of access required for a particular resource to function correctly.

For your infrastructure managed by Terraform, you can use **Pike**.

Pike is a tool that determines the minimum permissions required to run your Terraform code with one command.

It supports the big cloud providers (Azure, AWS, GCP).

Check it out: github(dot)com/JamesWoolfenden/pike

```
!brew tap jameswoolfenden/homebrew-tap
!brew install jameswoolfenden/tap/pike
```

```
pike scan -d .\terraform\
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "ec2:MonitorInstances",
            "ec2:UnmonitorInstances",
            "ec2:DescribeInstances",
            "ec2:DescribeTags",
            "ec2:DescribeInstanceAttribute",
            "ec2:DescribeVolumes",
            "ec2:DescribeInstanceTypes",
            "ec2:RunInstances",
            "ec2:DescribeInstanceCreditSpecifications",
            "ec2:StopInstances",
            "ec2:StartInstances",
            "ec2:ModifyInstanceAttribute",
            "ec2:TerminateInstances",
            "ec2:AuthorizeSecurityGroupIngress",
            "ec2:AuthorizeSecurityGroupEgress",
            "ec2:CreateSecurityGroup",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeAccountAttributes",
            "ec2:DescribeNetworkInterfaces",
            "ec2:DeleteSecurityGroup",
            "ec2:RevokeSecurityGroupEgress"
        ],
        "Resource": "*"
    }
}
```

12.1.4. Static Code Analysis Tool for IaC with `checkov`

Catch security problems in your Terraform code before they reach production.

This is easy to do with **checkov**.

checkov is a static code analysis tool for IaC with over 1000 policies.

As a CLI tool it is perfect for CI/CD pipelines.

```
!pip install checkov
!checkov --directory /user/path/to/iac/code

# check: CKV_AWS_21: "Ensure all data stored in the S3 bucket have versioning enable
#   FAILED for resource: aws_s3_bucket.customer
#   File: /tf/tf.json:0-0
#   Guide: https://docs.prismacloud.io/en/enterprise-edition/policy-reference/aws-p
```

13. Testing in Python

13.1. Testing

13.1.1. Pytest on steroids: Parallelize your test with `pytest-xdist`

Your test suite takes a very long time to run in Python?

With `pytest-xdist` installed, you can run your tests in parallel.

Specify the number of CPUs you want to use with `--numprocesses`.

This allows you to speed up your test executions.

```
!pip install pytest-xdist
```

```
!pytest --numprocesses 4
```

13.1.2. Shuffle the order of your tests with `pytest-randomly`

Sometimes it is interesting to run your test cases in a random order to check if there is any test case order dependency.

By running test cases in a random order, you can check if any test cases are dependent on the order of execution.

Skip to main content

To do that in Python, try `pytest-randomly`.

`pytest-randomly` is a pytest plugin to randomly shuffle the order of your tests.

It can be helpful in addition to other test strategies.

- The output tells you which random seed was used.
- If you want to use that seed again, use the flag `--randomly-seed`.

```
!pip install pytest-randomly
```

```
!pytest
```

13.1.3. Get Test coverage with `pytest-cov`

Do you want to measure the test coverage of your code in Python?

Try `pytest-cov`.

`pytest-cov` is a pytest plugin producing test coverage reports for you.

You can see how many statements in your code are covered in a nicely generated report.

Below you can see an example of how to use `pytest-cov`.

- With the –**cov** flag, you set the path to the module or package you want to measure coverage for.

-You can also specify the minimum required test coverage percentage using the `--cov-fail-under` flag.

```
!pip install pytest-cov
```

```
!pytest --cov=src --cov-fail-under=90
```

Skip to main content

```
-------------------- coverage: ... ----------------------
Name                  Stmts   Miss  Cover
---------------------------------------
src/__init__              2      0   100%
src/module1.py          257     13    94%
src/module2.py          100      0   100%
---------------------------------------
TOTAL                   359     13    97%
```

13.1.4. Test your plots with `pytest-mpl`

How to test your plots in Python?

Nowadays, you can test everything.

Functions, classes, websites, …

But how to make sure to check if your plots are correctly generated in Python?

Try `pytest-mpl`!

`pytest-mpl` is a Pytest plugin for testing plots created using Matplotlib.

It allows you to compare the output of your Matplotlib plots with expected results by automatically saving them as images and comparing them with pre-saved “baseline” images using image diffing techniques.

Below, you can see an example of how to use `pytest-mpl`.

- You have to mark the function where you want to compare images with `@pytest.mark.mpl_image_compare`.
- Provide the `--mpl-generate-path` option with the name of the directory where the baseline images should be saved.
- To test if the images are the same, provide the `--mpl` option.

```
!pip install pytest-mpl
```

Skip to main content

```
# testfile.py
import pytest
import matplotlib.pyplot as plt

@pytest.mark.mpl_image_compare()
def test_plotting_line():
    fig = plt.figure()
    plt.plot([1,2,3,4,5,6,7,8])
    plt.xlabel('X Axis')
    plt.ylabel('Y Axis')

    return fig
```

```
!pytest -k test_plotting_line --mpl-generate-path=baseline
```

```
!pytest --mpl
```

13.1.5. Instantly show errors in your Test Suite

When you run your tests with `pytest`, it will run all test cases and show you the results at the end.

But you don't want to wait until the end to see if some tests failed. You want to see the failed tests instantly.

`pytest-instafail` is a plugin which shows failures immediately instead of waiting until the end.

See below for an example of how to install and use it.

```
!pip install pytest-instafail
```

```
!pytest --instafail
```

13.1.6. Limit pytest's output to a minimum

Do you want to reduce pytest's chatty output?

Skip to main content

pytest-tldr is a pytest plugin to limit the output to the most important things.

A nice plugin if you don't want to be annoyed by pytest's default output.

```
!pip install pytest-tldr
```

```
!pytest -v # -v for detailed but clean output
```

13.1.7. Property-based Testing with `hypothesis`

Looking for a smarter way to test your Python code?

Use `hypothesis`.

With `hypothesis`, you define properties your code should uphold, and it generates diverse test cases, uncovering edge cases and unexpected bugs.

I encourage you to look into their documentation since it's really upgrading your testing game.

```
!pip install hypothesis
```

```
from hypothesis import given, strategies as st

@given(st.integers(), st.integers())
def test_addition_commutative(a, b):
    assert a + b == b + a
```

13.1.8. Mocking Dependencies with `pytest-mock`

Testing is an essential part of Software projects.

Especially unit testing, where you test the smallest piece of code that can be isolated.

They should be independent, and fast & cheap to execute.

Skip to main content

Here's where mocking comes into play.

Mocking allows you to replace dependencies and real objects with fake ones which mimic the real behavior.

So, you don't have to rely on the availability of your API, or ask for permission to interact with a database, but you can test your functions isolated and independently.

In Python, you can perform mocking with `pytest-mock`, a wrapper around the built-in mock functionality of Python.

See the example below, where we mock the file removal functionality. We can test it without deleting a file from the disk.

```
import os

class UnixFS:
    @staticmethod
    def rm(filename):
        os.remove(filename)
def test_unix_fs(mocker):
    mocker.patch('os.remove')
    UnixFS.rm('file')
    os.remove.assert_called_once_with('file')
```

13.1.9. Freeze Datetime Module For Testing with `freezegun`

When you want to test functions with datetime,

Consider using `freezegun` for Python.

`freezegun` mocks the datetime module which makes testing deterministic.

See below how we can specify a date and freeze the return value of datetime.datetime.now().

```
!pip install freezegun
```

```
from freezegun import freeze_time
import datetime

@freeze_time("2015-02-20")
def test():
    assert datetime.datetime.now() == datetime.datetime(2015, 2, 20)
```

13.1.10. Mock AWS Services with `moto`

If you have worked with AWS Services and Python,

you know how difficult testing your code can be.

With `moto`, you can easily mock out AWS Services and write your tests without headaches.t

Note: Not all services are covered, so check out the implementation coverage in their repository.

```
import boto3
from moto import mock_aws

class MyModel:
    def __init__(self, name, value):
        self.name = name
        self.value = value

    def save(self):
        s3 = boto3.client("s3", region_name="us-east-1")
        s3.put_object(Bucket="mybucket", Key=self.name, Body=self.value)

@mock_aws
def test_my_model_save():
    conn = boto3.resource("s3", region_name="us-east-1")
    conn.create_bucket(Bucket="mybucket")
    model_instance = MyModel("test", "testtest")
    model_instance.save()
    body = conn.Object("mybucket", "test").get()["Body"].read().decode("utf-8")
    assert body == "testtest"
```

13.1.11. Clean Output Diff with `pytest-clarity`

Improving the output of pytest with one plugin?

Skip to main content

It brings a coloured diff output which is much cleaner than pytest's standard output.

```
!pip install pytest-clarity
```

```
!pytest -vv --diff-width=60
```

13.1.12. Using Environment Variables with `pytest-env`

How to use environment variables when testing your code?

It's important to not mix up your test environment and local environment, especially when you want to define environment variables specifically for your tests.

For this case, use `pytest-env`.

`pytest-env` is a pytest plugin to define your environment variables in a pytest.ini file.

Those variables will be isolated from the local environment, perfect for writing and running tests.

```
!pip install pytest-env
```

```
# pytest.ini
[pytest]
env =
    API_KEY=example-key
    API_ENDPOINT=https://example.endpoint.net

# test_example.py
import os

def test_load_env_vars():
    assert os.environ["API_KEY"] == "example-key"
    assert os.environ["API_ENDPOINT"] == https://example.endpoint.net"
```

13.1.13. Mock your File system with `pyfakefs`

How do you write tests that interact with your filesystem?

Skip to main content

This can be tricky, as you don’t want to touch your real disc.

For Python and pytest you can use `pyfakefs`.

`pyfakefs` is a pytest plugin that provides an empty in-memory filesystem at each test start.

This allows you to test your file system interactions without breaking anything.

It comes with the fs fixture, which you can use right after installation.

```
!pip install pyfakefs
```

```
import os

def test_fakefs(fs):
    fs.create_file("/var/data/xx1.txt")
    assert os.path.exists("/var/data/xx1.txt")
```

14. SQL Tips & Tricks

14.1. SQL Tips & Tricks

14.1.1. `QUALIFY` Statement For Cleaner Queries

One underrated SQL command: `QUALIFY`.

With `QUALIFY`, you can filter the results of a window function like `RANK()` without needing another `SELECT` statement.

See the example below where we want to get the 3rd highest earner in every department.

```
# With QUALIFY
SELECT
  employee_id,
  department_id,
  salary
  RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS ranking
FROM
  employees
QUALIFY
  ranking = 3;

# Without QUALIFY
SELECT *
FROM (
  SELECT
    employee_id,
    department_id,
    salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS ranking
  FROM
    employees
) AS rank_employees
WHERE
  ranking = 3;
```

15. Miscellaneous

15.1. Miscellaneous

15.1.1. Must-Have VSCode Extensions for Python

Add these 8 must-have VSCode extensions when you work with Python to boost your productivity.

1. **Python Indent**: Makes sure your indentation is correct with every Enter you make
2. **Pylance**: A no-brainer. Includes many benefits like Parameter suggestions, Code navigation, Signature help, and many more. Microsoft declared it as the default language server for Python.
3. **GitLens**: Provides powerful features for your Git experience like seeing when a specific line was committed by whom in which pull request with which commit message. And much more. A must-have!

4. **Jupyter**: Notebook supports and allows any Python environment to be used as a Jupyter kernel.
5. **AREPL for Python**: Automatically evaluates Python code in real time as you type. Displays variables and errors in a readable way. Only works for Python >= 3.7.
6. **Python Path**: Helps you generate internal import statements in a Python project.
7. **Python Test Explorer**: Shows a test explorer without effort instead of going through the output from the terminal.
8. **autoDocstring**: Quickly generate docstrings for your functions

15.1.2. Project Setup from Templates with `cookiecutter`

Are you starting a new Data Science Project?

And go through the hassle of setting up the project structure?

Try `cookiecutter`.

`cookiecutter` is a command-line tool to create projects from templates.

This allows you to save time and have a standardized project structure.

There are tons of `cookiecutter` projects on GitHub you can use.

Say goodbye to tedious project setup.

Link to `cookiecutter` repository: https://github.com/cookiecutter/cookiecutter

Data Science `cookiecutter`: https://github.com/drivendata/cookiecutter-data-science

15.1.3. Project Scaffolding with `smol-developer`

Do you want something like create-react-app, but for anything?

Try `smol-developer` from smol-ai.

`smol-developer` scaffolds an entire codebase, based on a Markdown file with your specifications.

Skip to main content

You describe, what kind of application you want to develop and it will create the necessary boilerplate code.

Link to Repository: https://github.com/smol-ai/developer

15.1.4. Well Commits with `commitizen`

Clear and standardized commit messages are important.

But it's not always easy to have a standardized way in teams.

With `commitizen`, you will get a release management tool designed for teams.

It helps you define committing rules, bump project versions and create a changelog.

It makes your life easier by enforcing writing descriptive commits.

15.1.5. Docker Best Practice: Use `.dockerignore`

One Docker Tip:

Use a `.dockerignore` file to avoid adding unnecessary files to the image.

This will definitely reduce your Docker image size, but also more safe.

```
# .dockerignore
.git
.cache
*.md
!README*.md
README-secret.md
.env
```