Black, isort, bandit and other tools – formatting and static analysis of python code

The lazy man’s approach to assuring Python code quality

Pipelines

What are they and why do we need them

Automating stuff? Pipelines to the rescue

When we want to care about our Python code quality, we usually want to care about things like formatting, consistent import patterns, security and keeping our standars up to date. If we want to do that in our repo/in the cloud automatically, we can use pipelines.

Pipelines are simply a set of steps that constitute our **CI/CD process.**

It's more or less just a piece of code that does some steps for us. Usually pipelines are defined as a yaml file that definies what steps/actions we want to take as a part of our CI/CD process, meaning analysing, checking for quality, formatting of our code and building/deploying it.

In this presentation I'd like to Focus on the steps related to automation of the quality assurance process of developing Python apps.

Most commonly known tools for this in the cloud include: **GitHub Actions, GitLab CI/CD, Bitbucket Pipelines, CircleCI, Azure DevOps.**

Usually these are the things that fire up when we for example create a merge/pull request, push some code to the repo, merge one branch into the other. They trigger various checks, builds, tests and what not.

The flow is like so:
Trigger is received (eg. Branch is pushed to the repo) -> pipeline is fired -> various checks are made -> based on that pipeline can fail or succeed

Other than pipelines being there up in the cloud, I consider some parts of them an integral part of local development too. Mostly the parts related to the stuff about quality control.

What things to include?

Quality assurance of Python code

Handy little tools that automatically make our code better

What makes a good code?

Nowadays the trend in Python is to take care about certain things that while not crucial, over time contribute to the project's quality, readability and maintanability.

On a high level, in my book, any piece of Python code can use some of:

1. Consistent formatting
2. Ordered imports that are split by sections
3. Abolute imports
4. Usage of modern standards that are compliant to latest standards
5. Lack of unused imports and variables
6. Security/vulnerability scans

Black

Formatting? Why you should get black'ed.

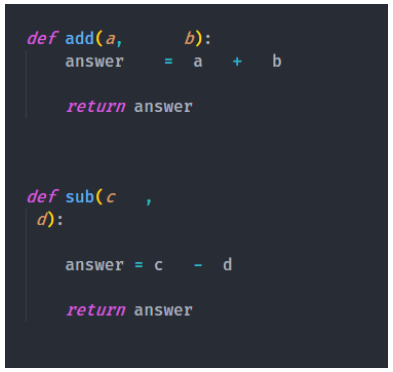
Few words on formatting and black

More often than not in projects that are not so automated and could use some of dem good tooling, you can find people in the pull requests arguing which formatting is better. How to change the formatting? Which one is better? Which one is more pep8 compliant?

It can be a nightmare that is as counter productive as it gets.

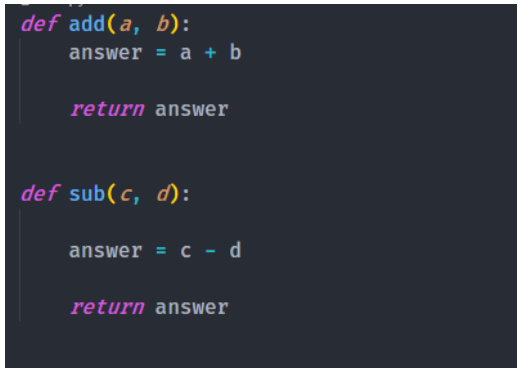
To get us rid of such problems and have it handled for us we use black in Python. Black is a code formatter that, well, just formats the code for you. You can make black automatically format your code before you commit. This way you can prevent any kind of arguments about pep8 and code formatting preferences of reviewers/authors, making the whole project have consistent formatting pattern, making it easier to read and so on. The easier code is to read, the better. It's the lazy man approach. If you know what to expect, you won't be surprised. The less you have to take care of, the better.

This:



```
def add(a,      b):
    answer    =  a   +   b

    return answer


def sub(c   ,
 d):

    answer = c   -  d

    return answer
```

Gets turned into this:



```
def add(a, b):
    answer = a + b

    return answer


def sub(c, d):

    answer = c - d

    return answer
```

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

' vs "

One thing worth noting is the fact that Python as a Language allows for the usage of both ' and " to mark strings. Black by default prefers double quotes over single. Why? Readability, usage of single quote in English language and the need to escape it everytime we use it inside our strings, it’s harder to mistake with ` sign.

So on so forth. One may argue here, I stand united with the double quote crowd as IMO it’s the better approach. Readability is king.

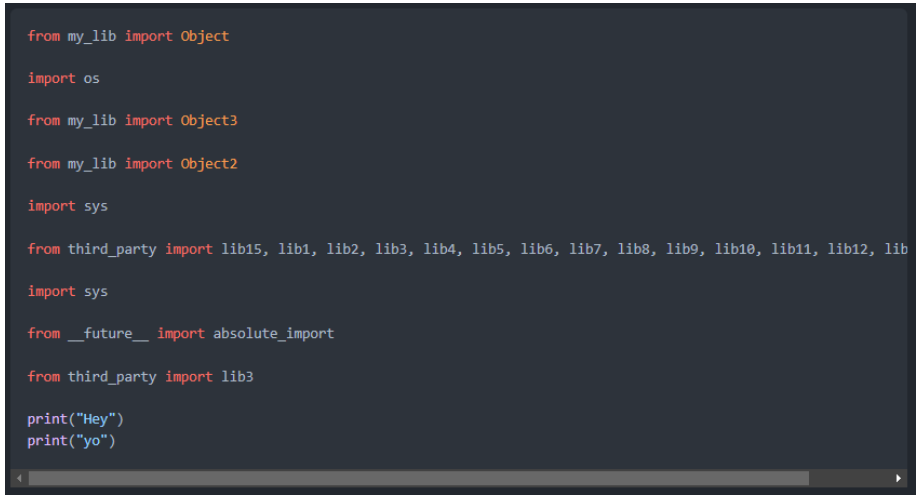
Isort

Have ya heard about imports sorting? It makes sense

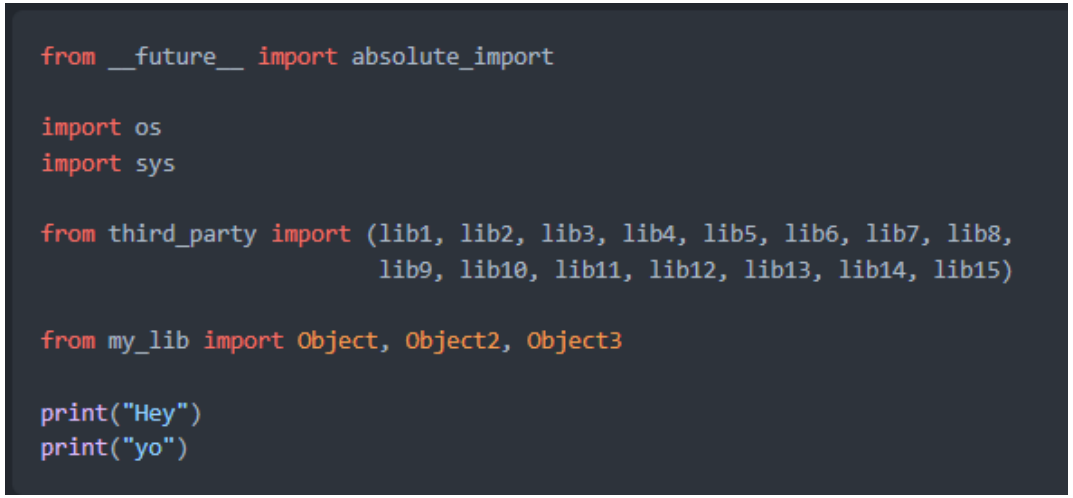
Why you should sort your imports properly

The bigger the project we work on, usually the more stuff we import from other pieces of the code.

As time goes by, these imports can become messy. It's often the case. Isort is something that helps us with that by optimising our imports, sorting them properly, alphabetically, grouping them in sections and so on. I know this can look like a minor thing, but it's these minor things that overall add to general code quality. Now look at the images below, the left one is before isort, right one is after it. Which one is more readable to you?



```
from my_lib import Object

import os

from my_lib import Object3

from my_lib import Object2

import sys

from third_party import lib15, lib1, lib2, lib3, lib4, lib5, lib6, lib7, lib8, lib9, lib10, lib11, lib12, lib

import sys

from __future__ import absolute_import

from third_party import lib3

print("Hey")
print("yo")
```



```
from __future__ import absolute_import

import os
import sys

from third_party import (lib1, lib2, lib3, lib4, lib5, lib6, lib7, lib8,
                         lib9, lib10, lib11, lib12, lib13, lib14, lib15)

from my_lib import Object, Object2, Object3

print("Hey")
print("yo")
```

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

Absolufy-imports

Handy little tools that automatically make our code better

What makes a good code?

The new standard is to have absolute imports. Why that is you can read on your own. There were multiple debates regarding that, the result of which is: when you can prefer absolute imports. They make for less ambiguity and provide clearer distinction of what we are really using, from which package.

We also have a tool for that which is absolufy-imports. This tool is especially usefull when dealing with older projects where you might need to fix the imports in a lot of files to fit the new convention. This tool does that for you.

This:

```
from .campaign_auto_closure_base import SomeClass
from .event_job_base import AnotherClass
```

Gets turned into this:

```
from em.jobs.notifications.some_important_file import SomeClass
from em.jobs.notifications.another_important_File import AnotherClass
```

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

Bandit

Static analysis of our code for potential security threads.

Why sometimes you need a bandit in your life

When we write our code we should have security in mind. Unless you sometimes want to make your company vulnerable to potentially losing millions. I'm going overboard with this example, but still. Security is important.

Somehow we can make mistakes simple because of forgetfulness and negligence that could have been prevented otherwise.
To remind us of this there are various tool that you can use.

Among them is bandit. Bandit is a static analysis tool that scans your code for potentially unsafe fragments of code and warns you about them.

When you run bandit against your code you'll get a report like this and a list of where in code the potential problems are.

```
Code scanned:
    Total lines of code: 52868
    Total lines skipped (#nosec): 0

Run metrics:
    Total issues (by severity):
        Undefined: 0
        Low: 105
        Medium: 38
        High: 7

Total issues (by confidence):
    Undefined: 0
    Low: 15
    Medium: 18
    High: 117
```

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

autoflake

The less you have...

Reducing waste

Sometimes it so happens that we may have unused import statements in our code or unused variables. Happens to the best.

In order to automatically take care of those we may want to include autoflake in our projects.

It’s a tool that simply takes care of that – removing unused imports and variables.

No magic here.

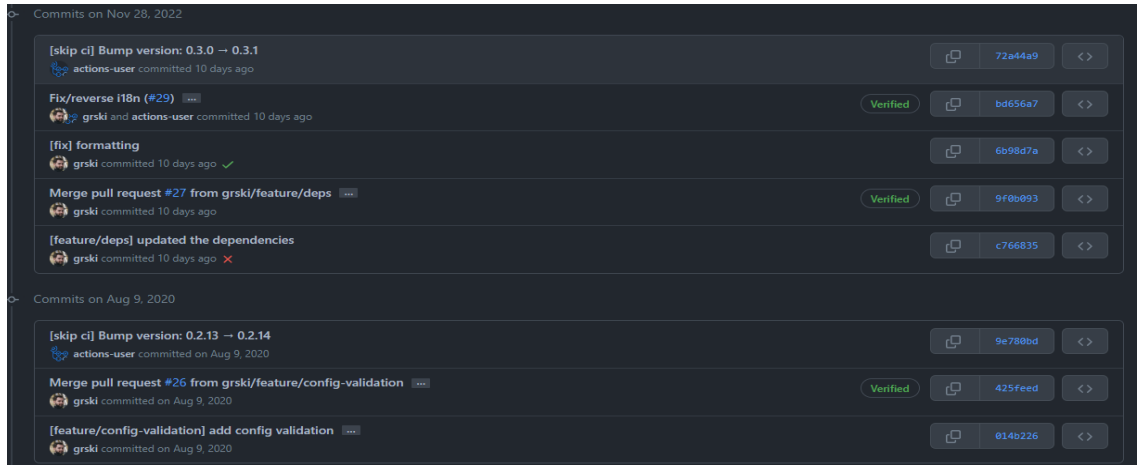
bumpversion

Automatic semantic versioning

Human readable versions that make sense

There's this thing we call semantic versioning or semver. It's a convention that tells us to version our code according to the following pattern:
MAJOR.MINOR.PATCH
For example: v0.2.12

Major piece is incremented when we go for major rollouts that change A LOT.
Minor piece is incremented when we do normal releases eg with bigger features.
Patch is something we use for smaller features, patches, fixes etc. This one grows the fastest.
In order for us to not have to manage it manually, we have a tool called bumpversion. It updates the version, creates a commit with the changes, creates a git tag and so on, all automatically. It's a neat little piece of tooling to have in you CI/CD.

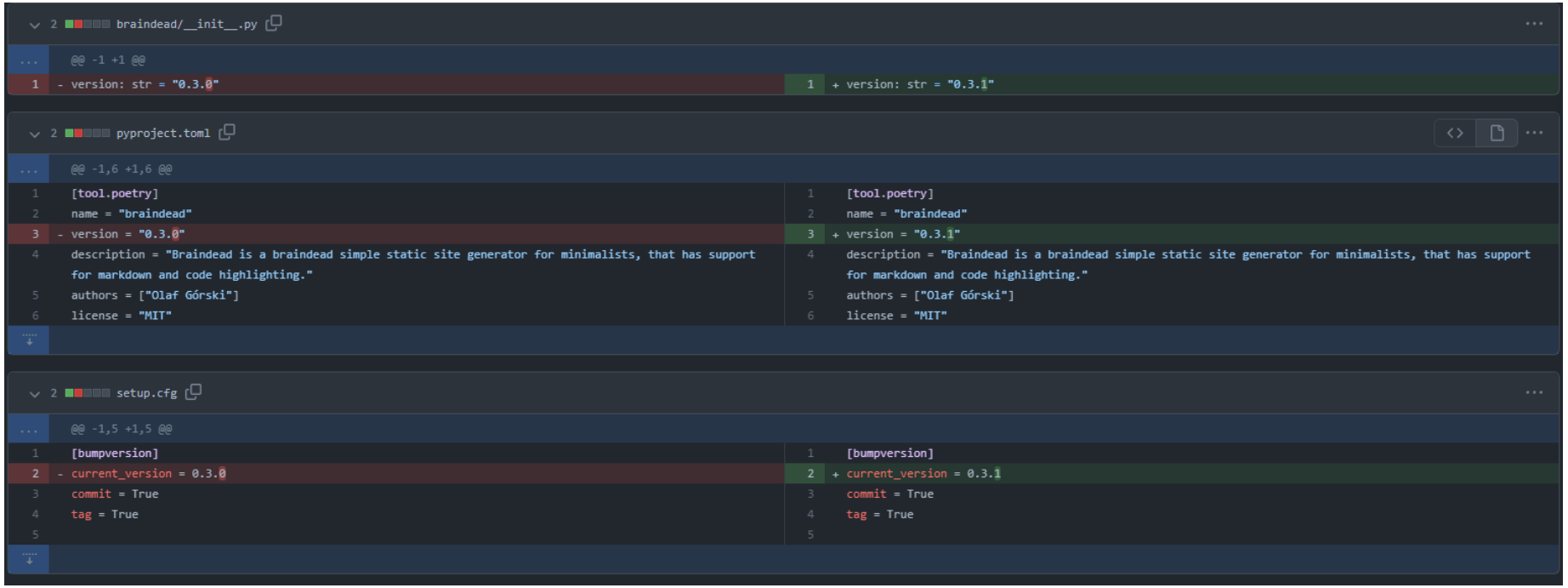


Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

bumpversion

Automatic semantic versioning

Automatically update the version in the files too.



Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

Do we run all of these by hand?

No. We want to be lazy.

Git hooks

Automate boring tasks

Git hooks and pre-commit

If you want to make all of this happen automatically, you can create git hooks that are fired eg. When you commit or before the commit.
One way is to just **create .pre-commit file and put it in your .git** folder and leverage eg. **Makefile** or use something like **pre-commit** tool.

It’s a nice handy tool that handles this for you. You need to install it and create config for it to tell it which things to do before the commit.

No magic here.

I’ll let you google the details yourself ☺

Git hooks

Automate boring tasks

Example Makefile

```
PATH  := $(PATH)
SHELL := /bin/bash

flake:
    flake8 -v ./

isort:
    isort --check-only --diff ./

isort-inplace:
    isort ./

bandit:
    bandit -x './styles/*' -r ./

black:
    black --check --line-length 120 --exclude "/(\.eggs|\.git|\.hg|\.mypy _cache|\.nox|\.tox|\.venv|_build|buck- out|build|dist|migrations|node_modules)/" ./

linters:
    make flake
    make isort
    make bandit
    make black

bumpversion:
    bumpversion --message '[skip ci] Bump version: {current_version} → {new_version}' --list --verbose $(part)

black-inplace:
    black --line-length 120 --exclude "/(\.eggs|\.git|\.hg|\.mypy _cache|\.nox|\.tox|\.venv|_build|buck- out|build|dist|migrations|node_modules)/" ./

autoflake-inplace:
    autoflake --remove-all-unused-imports --in-place --remove-unused-variables -r --exclude "styles" ./

format-inplace:
    make black-inplace
    make autoflake-inplace
    make isort-inplace
```

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

Serious self-righteous stuff aside,
here are some memes I enjoy to lighten up the mood

Nie najtaniej, ale jako tako. It ain't much, but it's honest work.

Feedback? Questions?AMA.
Let’s get to know each other.

Thank you!