Python Formatter

This document is a draft project report for my B.Sc

Table of Contents

Problem stating

Almost every language has a set of standards that developers advocate. These standards are essentially a set of rules on how you should name your files, variables, classes, and even non-naming issues such as line length.

Coding standards are designed to try and make the code you write as maintainable as possible, when developers work as a team on a project, having each member writing their code based on their own style will make the code unreadable, and unmaintainable, even when working as a single developer, a coding convention will make the code look consistent and easier to read.

Most projects already have a coding convention, but how do we enforce them? Code reviews are certainly not good enough since they take time and energy, automating this task can achieve maintaining a well formatted code with minimal extra work.

A software for enforcing a coding style is needed in order to save the developers time and effort.

Our project automates formatting the code, the software accepts python code as an input, and outputs the code formatted according to the rules of PEP8 [1], with emphasis on keeping it as customizable as it can be.

2 Project Goals and Specification

The software is a tool that can be used to help/enforce a specific guideline, the default guidelines for the software are the ones mentioned in PEP8 but can be customized to match the developer's need.

When running the software, the user should provide a path of a specific python file or a directory and it formats the python file/python files in the directory and subdirectories to match the rules, optional arguments can be also provided to override the default rules.

The Python formatter we want to implement must serve as many Python developers as possible, it should be dynamic, customizable, and can be integrated easily with source control software.

Specifications:

1. Input:
    a. A path (or paths) of a python file(s).
    b. A path(s) of a directory/directories that contains python files, the formatter will search through the directory and the subdirectories for Python files and other files from a specific format if the user provided additional suffixes.
2. Allowed customizations:
    a. Max length of line: The default value is 88.
    This option will restrict the maximum length line and will rewrite a long line to a shorter one using multiple lines.
    Note that in some cases this might lead to an error since it is not always possible, for example when an identifier such as a variable or a function name is longer than the default/provided maximum length, in such cases, the formatter will raise a NoSolutionError.
    b. Number of vertical white lines between nested functions: The default value is 1.
    This option allows specifying the number of vertical white lines between nested functions; these nested functions could be class functions, or functions defined within other functions (aka nested).
    c. Number of vertical white lines between global functions and classes: The default value is 2.
    This option allows specifying the number of vertical white lines between global/module scope functions and classes.
    d. Whether spaces should be placed between arguments and their default values: The default value is True.
    This option is relevant for functions and classes with default values, and for keywords used in a function or a class call.
    e. Whether to allow multiple imports on the same line: The default value is False.
    This option will allow the user to import multiple imports in one liner.
    f. Whether the system should transform the code or to indicate if the code is written according to the guidelines: The default value is false.
    This option is useful when using the software in a CI/CD cycle, more on that in Integration with github section.

3. Providing the customizations rules:

    a. Through command line arguments.

       All the supported command line arguments can be displayed using the -h or –help flag with a small description.

    b. Through text configuration files.
       The formatter has a template file called conf.txt which holds the names of the supported arguments and their default values, the user could change these default values or provide a similar configurations file through the command line arguments using -cfg <path> or the longer name –configuration <path>.
    c. The command line arguments take the priority over the configuration files, the idea behind this decision is to allow the users to experiment with the supported arguments faster.
4. Integration with source control software.
   A very powerful feature that could help group of users to work with a single format, this feature is available using the –check-only argument, which was mentioned in 2.f, the idea behind this is that a shell runs the software on the committed source code and indicate whether a code is properly formatted by the return code.

   The following table describes the return values we support:

Return code	Description
0	Success – The code is well formatted.
1	Failure – The code is not written by the specified guidelines
2	NoSolutionError

Table 1

3 preliminaries

3.1 PEP 8

PEP8 is a document which provides coding conventions for the Python code comprising the standard library in the main Python distribution.

This document and PEP 257 (Docstring Conventions) were adapted from Guido's – Python creator - original Python Style Guide essay, with some additions from Python contributors.

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

3.2 Abstract Syntax Tree (AST)

An Abstract Syntax Tree, or AST, is a tree representation of the source code of a computer program that conveys the structure of the source code. Each node in the syntax tree represents a construct occurring in the source code.

During conversion to its abstract syntax tree, only the structural and content-related details of the source code are preserved, and any additional details are discarded. Information that are preserved, and are vital to the syntax tree purpose, are:

- Variable types, and location of each variable declaration
- Order and definition of executable statements
- Left and right components of binary operations
- Identifiers and their assigned values

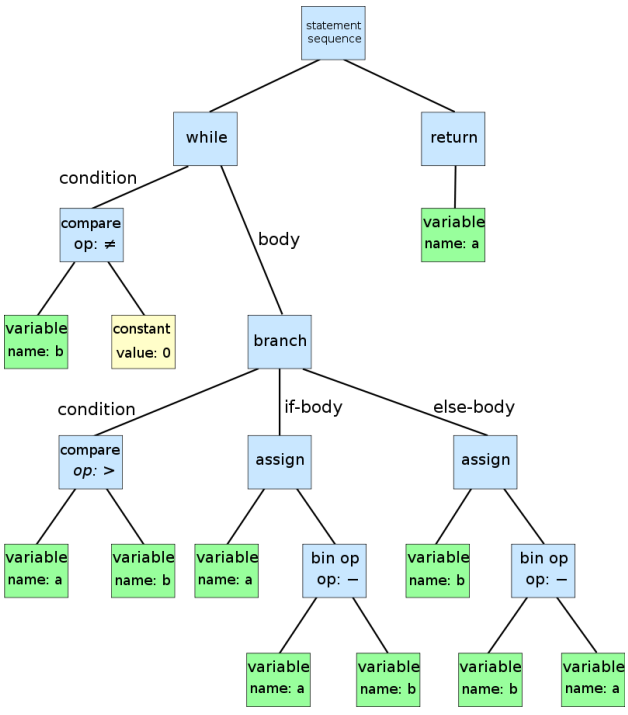


Figure 1

3.3 Important Python features

Unlike C, C++, and many other languages, Python is interpreted language and not compiled.

An interpreted language is a programming language which are generally interpreted, without compiling a program into machine instructions. It is one where the instructions are not directly executed by the target machine, but instead read and executed by some other program.

Being read and executed by some other program means that the code will run line by line, and being also a dynamic language, brings powerful tools to Python, during run time, at any given point, a programmer can access the stack, can extend the code, and look up at any instance of an object and access its attributes and even add new ones. For example:

```
class person:
    def __init__(self, id):
        self.id = id
new_person = person(123456789)
# new_person is an instance of  person which has only id field
setattr(new_person, 'name', 'David')
# new_person has a new field called name and its value is set to
# David, in addition to id field
```

At the same time, Python offers the developer to check whether a given field (or attribute) exists

```
# The following functions returns the value of last_name of
# new_person if existed, returns None otherwise
getattr(new_person, 'last_name', None)
```

These python features will prove to be very useful later and will make Python the perfect candidate for this kind of project.

4 Implementation

The main idea behind the implementation is first, to understand the syntax, and write a new code with the same syntax.

To understand the code, we had to research the grammar of Python, and understand its lexical elements so we can extract the Abstract Syntax Tree of the code, after extracting the ast, we traverse it and write the code from the start.

The mechanism itself is very simple:

For each file:

1) Extract the AST.
2) Traverse the AST and rebuild the code by printing it line by line to a temporary file.
3) Replace the original file's content with the temporary's file content.

4.1 Setting up the configurations

A special class Conf is dedicated for the configurations of the formatter.

Conf is responsible for:

1. setting up the initial configurations according to the conf.txt file which is in the project files and contains all the configurations and their default value
2. Parse the command line arguments and adjust the configurations accordingly, if an additional configuration file is provided, it will redo step 1 with the external configuration file.
3. Conf is also responsible for printing the display message correctly (including the indentation).

4.2 Searching for the files

After configuring the formatter with the needed parameters, _search.py module is responsible for traversing the given directories and their subdirectories with *walk()* method, this method gathers all permitted files to be formatter and saves the file's paths in a list.

The files that get appended to the list are either python files or any other files with a suffix that were provided in the configurations stage.

4.3 ast module

According to the Python documentation the ast module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

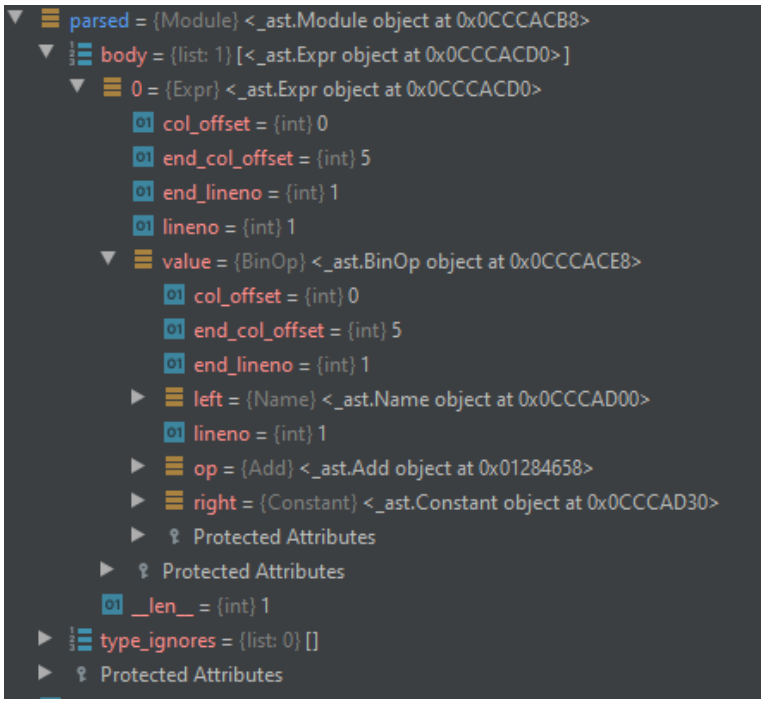
This module contains an abstract AST class which contains useful data about each node in the ast, other than the AST class it also provides helper functions such as parse(), both of these functionalities will be explained furtherly below.

4.3.1 AST class

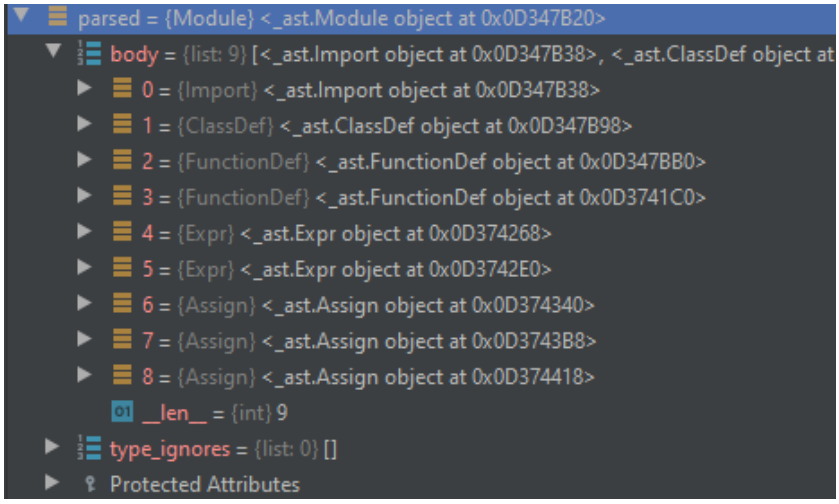
An AST represents each element in a code as an object. These are instances of the various subclasses of AST. For instance, the code a + 1 is a BinOp, with a Name on the left, a Num on the right, and an Add operator, BinOp is a subclass which contains three important fields, left which holds a node of a type Name in our example (which is also a subclass of the AST), an op; which describes the binary operation, in our example it's the plus sign, and the right field, which in the example above contains a Num which is another subclass of the AST.

4.3.2 parse() method

One of the useful methods it has is the parse() method, which parses a source code into an AST node. The following figure could give an idea on what the parse returns when calling on '*a + 5*'



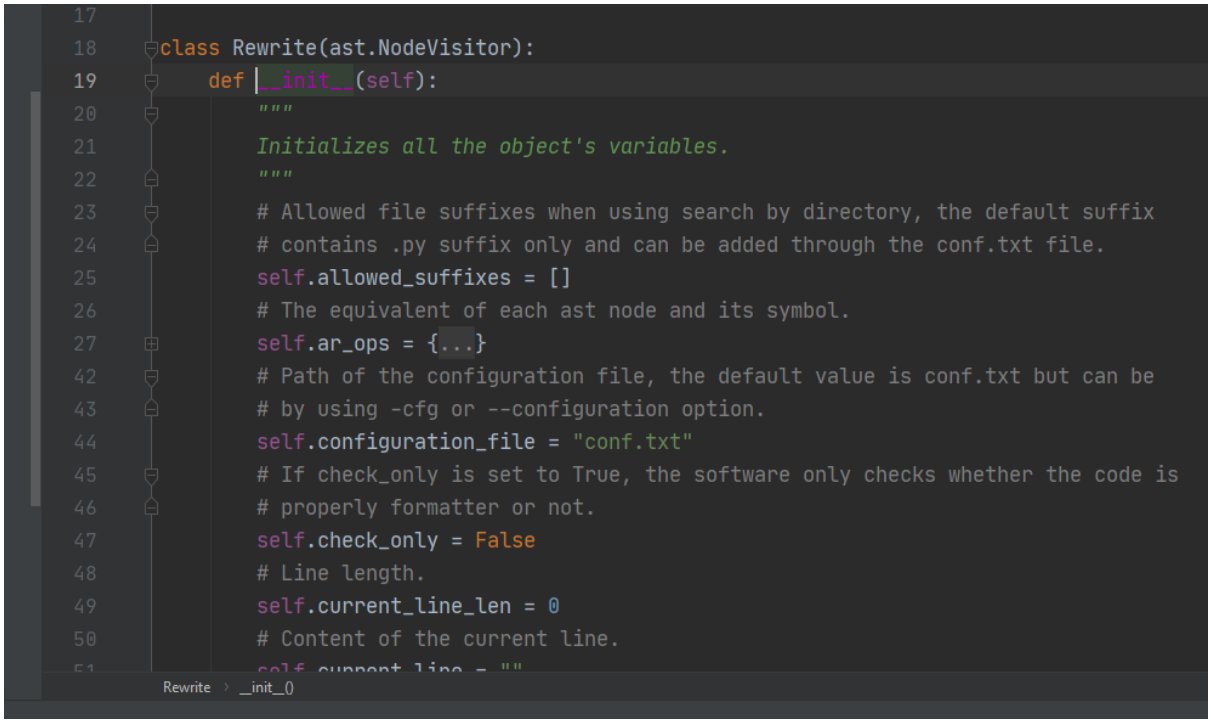
Another example of an AST on a fully function code, in the following figure the specifics of each node are not opened since it will make the picture way too big in height.



As you can see, the Abstract Syntax Tree of a Python code is ironically a list.

4.4 Extending the AST Class

To get the full potential of the AST class we had to create a first-level subclass of the AST which still beholds all the AST's subclasses. We had to make the class in so we can create a practical and dynamic tree traversal with our own methods and helper functions. Python also provides such class to inherit from which is the NodeVisitor class, the figure below shows the definition of our special class that inherits from the AST.



4.4.1 visit()method

The AST traverse is done using the method visit(), a clever way to traverse the way depending on the node type:

```python
def visit(self, node, new_line=True):
    """
    Visit a node, this overrides NodeVisitor visit method as we need to
    start a new line between each body element.
    """
    method = "visit_" + node.__class__.__name__
    # Get the visit_Class method, if not found, return generic_visit() method.
    visitor = getattr(self, method, self.generic_visit)
    logging.info(f"in visit(), visitor={visitor.__name__}")
    # Call the visitor function
    visitor_ = visitor(node)
    if new_line and not isinstance(node, ast.Module):
        self.new_line()
    return visitor_
```

This function gets the class name and put it into a variable with the visit_ prefix, for example, if the node is a function definition node, the method variable seen in the figure above will contain *"visit_functiondef"* after that, we use the built-in attribute "*getattr()*" which checks if a given class has the an attribute with the same name as the string, if yes, it returns the attribute, if not it returns *"self.generic_visit"* , the generic visit is also a method of the class for all the nodes that don't have a special function, it's needed in some cases, and what it does is that it traverses on the node fields and calls "*visit()*" on each node in order to continue the traversing recursively.

A simple, minimalistic example of a dedicated visit method for an AST subclass (a type of a node) is the visit_Name method, which handles all the Name nodes, these nodes contain identifier names, such as variable and function names.

```python
def visit_Name(self, node):
    """
    Implements name (identifiers).
    :param node: _ast.Name.
    :return: None.
    """
    logging.info(f"in visit_Name, node.id={node.id}")
    self.print(node.id)
```

4.4.2 Indentation

The indentation is saved into a special variable, which could be incremented and decremented by the special *__enter__* and *__exit__* Python functions, these functions are called when using Python's *with* statement.

When using "*with self*:" in python the previously mentioned methods gets called, each one is responsible for adding or removing 4's which describes the indentation at any point.

```
    with self:
        # Visit all the the nodes body block.
        for i, element in enumerate(node.body):
            if i + 1 != len(node.body):
                self.visit(element)
            else:
                self.visit(element, False)
    if node.orelse:
```

The figure above is an example of a usage, which is in an if statement, right before the block starts, *__start__()* is called, and right after the block ends, *__exit__()* gets called.

These are useful because they are easy to use, and easy to detect, instead of implementing special functions which will get "lost" in the code, they are visible, each to catch, and make the code much more readable.

4.4.1 print() method

The *print()* method is responsible for printing the correct syntax to the file. The way *print()* method works is that it gets two arguments (more actually, but two important ones), first the value, and second, whether by the end of printing the line there is a new line. In many cases, a single line will be composed of multiple *print()* calls. The formatter doesn't print these lines at the spot, but instead, the formatter has a *"current_line"* variable which has the content of the current line at each given moment, once the second argument, the *new_line* flag is set to True, it prints the content of *current_line* to the file and reinitiate it with an empty string.

When the string is empty, the first thing we do is add the indentation, after that we add what is needed to be printed and check whether we exceeded the maximum line, if we did, we initiate special helper values in order to get the formatter to understand that we have a line the exceeds the maximum length, we set a variable indicating that we are currently in a node that requires special handling, and calls *visit()* on the node that have called the first *print()* since the last line was printed, this way we could ensure that we will go back to the starting point, but this time, the formatter knows that it should break the nodes into as many lines as it can. If at one point we exceed the maximum line length, a NoSolutionError is raised, otherwise, we continue to finish the line and reset the helper variables we mentioned above.

```
def check_line(self):
    """
    Checks if the current line exceeded the max length, initializes the needed
    variables to start a new line and calls the main node that starts the line.
    If the node supports checking for long lines, it will be handled well, the
    program will go into an endless recursion.
    :return: None
    """
    if self.current_line_len > self.max_line:
        logging.warning("Line exceeded limit")
        self._init_values_for_long_line()
        self.visit(self.starting_new_line_node, new_line=False)
        self.starting_new_line_node.exceeds_maximum_length = False
        self.long_node = False
```

4.5 Integration with Github

The integration with github is done using the software's –check-only option and github actions.

4.5.1 –check-only Option

When using this option, the software returns one of the three codes (see Table 1), which the shell running the software could interpret the output.

The software does modify the original files, instead it keeps the temporarily created file and compares it to the original file, If the files are not identical, it appends the file name to the list of files that are not well formatted and continues to the next file. If the formatted list is not empty, the software prints the paths of the files that must be changed, and returns 1, if the software catches a NoSolutionError during the run, it stops executing and returns 2, if all the files were reformatted and proven to be identical to the original files, the return code will be 0.

4.5.2 GitHub Actions

GitHub Actions allows building continuous integration and continuous deployment pipelines for testing, releasing and deploying software without the use of third-party websites/platforms.

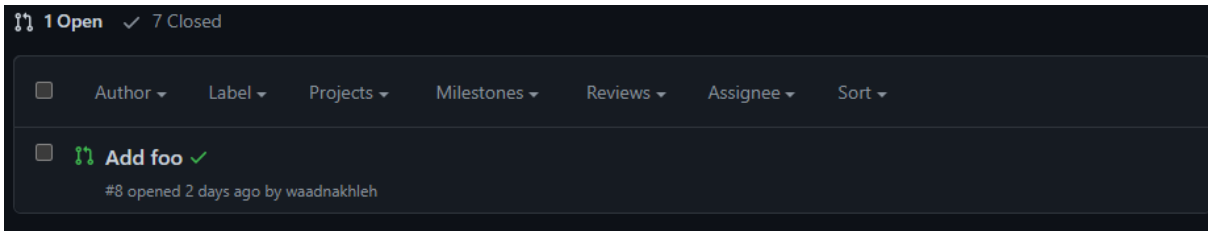
The idea is to whenever there is a specific event (such as a pull request, new commit, etc.) github servers can run shell commands on the repository, in our case, we would like to make github run the formatter with the check-only option and to report back the to the user, so the action will pass only if the formatter exists with 0 as a return code. This can be done using the special yml file, files that control the github actions. To make github run the formatter on any repo, the only thing is needed is to upload the formatter to a dedicated repository (could be public and accessed via different repositories and people) and to add the yml file.

The *.yml file should contain the following:

```
name: GitHub Actions
on: [push]
jobs:
  Run-Formatter:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v2
        with:
          python-version: '3.x' # Version range or exact version of a
# Python version to use, using SemVer's version range syntax
          architecture: 'x64' # optional x64 or x86. Defaults to x64 if
# not specified
```

```
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Checkout to pythonformatter
        uses: actions/checkout@master
        with:
          repository: waadnakhleh/pythonformatter  # Path of the formatter
# repository
          path: formatter
      - name: Execute
        run: |
          cd formatter
          python -m main -d ${{ github.workspace }} --check-only
      - run: echo "This job's status is ${{ job.status }}."
```

This code will add a green check indicating a pass if the formatter returns 0, otherwise it will add a red X indicating a failure



4.6 Testing

The Testing is done with pytest library, a framework which enables writing fully featured tests; unit and integration tests; pytest is widely popular, opensource, and easy to use.

The tests were divided into three categories:

4.6.1 Unit/Functionality tests

These tests usually test a specific node type, for example, a dedicated test for if statements, or even simpler types such as lists and assignments.

For each node type we have added a dedicated unit test to ensure a proper functionality for the formatter, and while these tests can be dedicated to a single node type, some types do not work without adding other nodes, this is due to the grammar limitations of the Python (and each other language as well), an example of this is an assignment, the grammar for the assignment as the left hand side and the right hand side of the assignment could be of multiple types, in order to minimize an integrated test within a unit test we have tried to keep a minimal usage of different nodes.

4.6.2 Integration tests

Integration tests are tests that include multiple units of the code combined to check whether the software works well together in overall.

We didn’t add many of integration tests since a huge amount of unit tests became also an integrated test. We took random opensource python scripts from the internet and added them randomly together in the integration tests.

4.6.3 Customizations test

Customizations tests ensures that the command line arguments and the configurations files work well, and they aim to test whether they work well alone and together since providing a configurations file and command line arguments together is also possible, in addition, there are tests to validate the command line arguments that can be provided multiple times such as –directory argument.

One of the useful methods it has is the parse() method, which parses a souce code into an AST node.

The AST node class is an abstract class, being the abstract class for all the AST node classes, each class describes a syntax, an example of an AST tree in the ast module:

5 Challenges

During the development stage, our team has faced many challenges that we needed to face.

It was extremely hard to understand the Python grammar and to look into a way to parse the code, luckily, we got the parse() method mentioned in the implementation section. But in order to understand the returned structure we needed to understand the AST class and how ASTs in python are structured, which are very different from the typical languages, there was no proper documentation, the one which exists at Python's official docs are not very useful since they give a brief explanation on some of the methods and classes and not all of them, and the ast module we used is not a popular module to be used by developers, its main purpose is to help programers to understand the Python grammar and not to use it for a Python formatter or any other software, thus information on the internet were extremely limited and we had many questions that we couldn't find online, some of the questions we got to know the answer by playing with the module by ourselves which took a good chunk of time, and some of the answers we got by directly asking on websites such as stackoverflow and reddit.

Keeping the lines limited to the maximum length was even a bigger challenge that contained many challenges within itself, as there were many soft spots and different approaches that each had its own advantages and disadvantages. One of the challenges we faced is our approach to the whole how-to-solve the issue, in a nested class, such as the function definition, it contains many other nodes inside it, but the function definition itself is the marks the first letter of the line, if somewhere in the middle we're visiting an argument node, we got to know exactly how much to go back until we get to the correct parent so the formatter will try to print once again the function definition but this time knowing that it exceeds the maximum line length, so it tried to break the lines accordingly, there are many different parameters that could contribute to how the line should be printed, and it took a loot of trial and error until we got to the right solution in our print() method.

The integration with Github took a lot of time given its simplistic idea behind it, we thought that activating the software from a remote shell when there is a push event would be a simpler task, but yet it took a lot of researching and understanding how github actions work and the syntax of the .yml files that control these actions to get to integrate our formatter with github. Instead of an estimated half a day it took about a week.

6 Future Features

6.1 Add support for code analysis:

The formatter could be extended to support futures beyond formatting the code, a possibility is to check for unused variables, duplicated code, assessing the possible bugs, e.g., the software could raise a warning when a list is passed to a function (which is a dangerous and not a good practice), in other words, detection of vulnerabilities and functional errors.

6.2 Add support for optimization:

Python is one of the few famous languages that guarantees that each line of code will be executed in the same order as the code, while this can be useful in some cases, many cases does not require this guarantee, adding optimization that change the code, on default settings it should be disabled but can be beneficial if the user decided to use it.

6.3 Add support for future Python versions

The software is built in a way that adding support for versions > Python3.8 should not be a hard task, since the ast module we mainly focused on does not change frequently, and when it does, only a few parts of it do.