# CS 846 - Final Project

*Comparative Analysis of AST (Abstract Syntax Tree)-based Tools in Python Code Formatting*

---

Daniel Phan

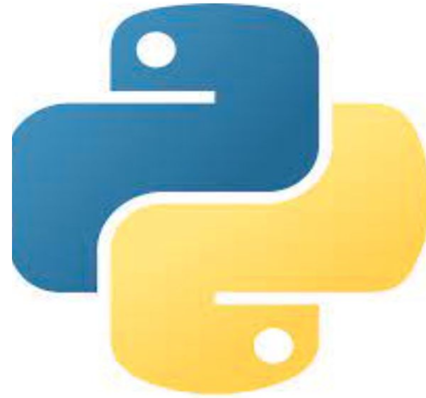
*daniel.phan@uwaterloo.ca*

# What is the Python programming language?

- “Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.”
- Fast growing programming language in the modern era
  - Machine Learning
  - Scientific Programming
  - Big Data Processing Engines
  - Support in various areas:
    - User Interface
    - Web Development
    - API Development
- Resources:
  - https://www.python.org/doc/essays/blurb/
  - https://en.wikipedia.org/wiki/Python_(programming_language)
  - https://developers.google.com/edu/python

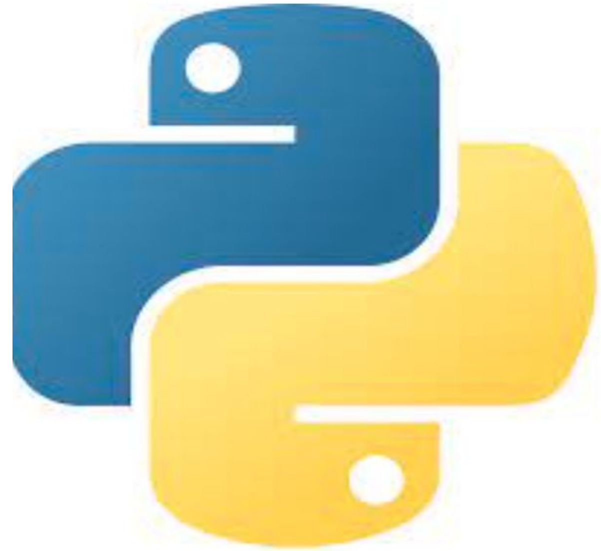
# Strengths of Python

- Highly readable, easy to use, and easy to pickup
- Wrapper of various functionalities
- Dynamic Typing System
  - Type system is abstracted away
  - Ability to write highly concise code (especially in research)
  - Highly suitable for scripting tasks
- Strong community support
- Rapid development and prototyping (easier for research purposes)
- Integrations with many other tools
- Suitable for scripting
- Resources: [https://www.researchgate.net/figure/Python-code-sequence-15_fig3_347776583](https://www.researchgate.net/figure/Python-code-sequence-15_fig3_347776583)



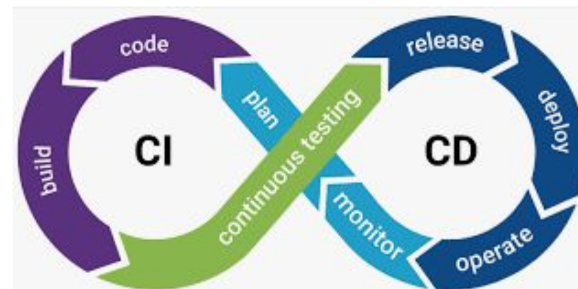
# Weaknesses of Python

- Performance
- Memory
- Concurrency
- Error prone in some cases
- Type checking
- Can be hard to maintain due to lack of standards
- Although can be used for general purposes, but tasks like front-end development is not well-supported
- Resources:
  - https://devguide.python.org/
  - https://docs.python.org/3/library/index.html



# Research Motivation

- Why formatting for code?
  - Improved readability
  - Improved maintainability
  - Improved developer productivity and facilitate collaboration
  - Easier for debugging and testing
  - Boost up the software development life cycle
  - Understand the intrinsic logic of the algorithms
  - Error prevention
- Adhering to a global stands
  - Automation
  - World-wide collaboration
  - Code-reuse across repos
  - Better code diffs across commits

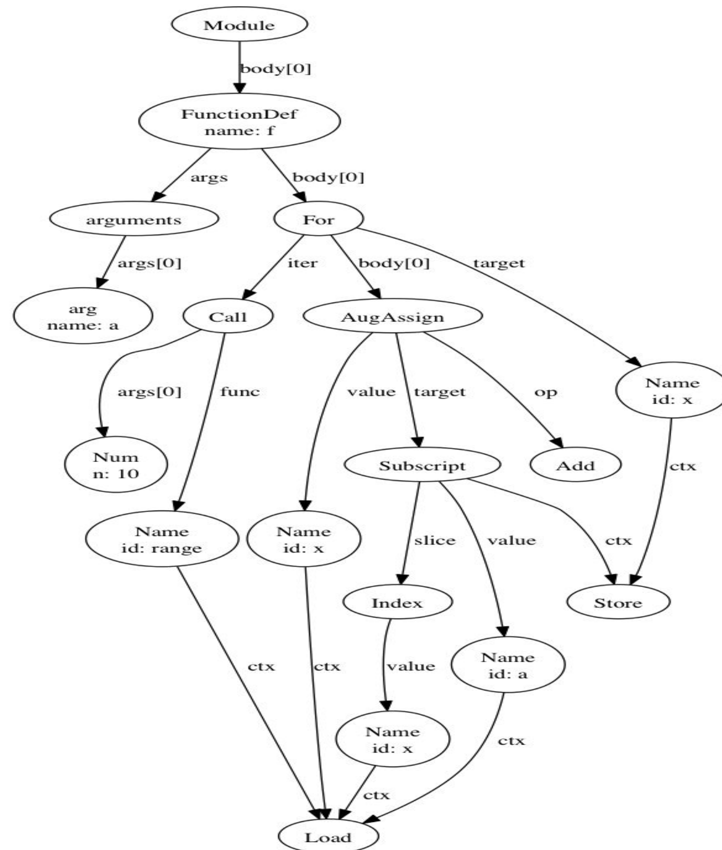


# What is an AST?

- ***Definition:*** "An ***Abstract Syntax Tree (AST)*** is a representation of the structure and meaning of a parse tree in computer science that eliminates extraneous nodes, focusing on the essential details of the derivation process."
- ***Resources:***
    - https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree#:~:text=Abstract%20Syntax%20Tree%20(AST)%20is,until%20all%20dependencies%20get%20connected.
    - https://en.wikipedia.org/wiki/Abstract_syntax_tree
    - https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38
    - https://docs.python.org/3/library/ast.html
    - Videos (by Dmitry Soshnikov)
        - https://www.youtube.com/watch?v=VZ5DJopq5JA&list=PLGNbPb3dQJ_6aPNnlBvXGyNMIDtNTqN5I&index=1
        - https://www.youtube.com/watch?v=VjvPnpzcJh4&list=PLGNbPb3dQJ_6aPNnlBvXGyNMIDtNTqN5I&index=2
        - https://www.youtube.com/watch?v=VKM1eLoN-gI&list=PLGNbPb3dQJ_6aPNnlBvXGyNMIDtNTqN5I&index=3

# What is an AST?

- Specifically, in Python
  - There is a specialized built-in module for AST parsing
- Multiple types of abstract nodes
  - Root nodes
  - Literals
  - Variables
  - Expressions
  - Subscripting
  - Statements
  - Imports
- Resources
  - [https://docs.python.org/3/library/ast.html](https://docs.python.org/3/library/ast.html)



# Example 1

```
print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'), indent=4))
```

```
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())],
                    keywords=[]),
                conversion=-1,
                format_spec=Constant(value='.3'))]))
```

# Example 2

```
print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
```

```
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
             value=Constant(value=2))])
```

# Example 3

```
print(ast.dump(ast.parse('a, *b = it'), indent=4))
```

```
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))],
    type_ignores=[])
```

# Example 4

```
print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
```

```
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

# Example 5

```
print(ast.dump(ast.parse('type Alias = int'), indent=4))
```

```
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[],
            value=Name(id='int', ctx=Load()))],
    type_ignores=[])
```

# Example 5

```
print(ast.dump(ast.parse("""
... match x:  case Point2D(0, 0):  case Point3D(x=0, y=0, z=0):
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))],
                        kwd_attrs=[],
                        kwd_patterns=[]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        patterns=[],
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

```
print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[

MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

# Example 6

```
print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) ->
'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(
                        arg='a',

annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1',
ctx=Load()),
                Name(id='decorator2',
ctx=Load())],

returns=Constant(value='return
annotation'),
            type_params=[])],
    type_ignores=[])
```

# What is a code formatter?

- **Input:** Code
- **Output:** Formatted Code
  - Predefined Rules | Added extra symbols | Removed extra spaces | Fully-automated
  - Work in real-time (for example, on file-save events or specific events)
  - Can be many more! In general, the formatting rules are highly customizable
- Resources
  - https://code.visualstudio.com/docs/python/formatting
  - https://github.com/life4/awesome-python-code-formatters
  - https://devblogs.microsoft.com/python/python-in-visual-studio-code-may-2018-release/

```
53  55
    56 +
54  57   def languages_breakdown(year):
55  58       if not os.path.exists(survey_csvname(year)):
56  59           download_survey(year)
57  60
58  61       print(f"Processing {year}")
59     -     data=pd.read_csv(survey_csvname(year), encoding='latin1')
    62 +     data = pd.read_csv(survey_csvname(year), encoding="latin1")
60  63
61  64       if year >= 2016:
62  65           # Languages are semicolon separated list in a single column
63     -         languages = data[questionNames[year]].str.split(';', expand=True)
    66 +         languages = data[questionNames[year]].str.split(";", expand=True)
```

# Existing Tools

- The 3 most powerful and widely used tools in the Python community
  - PEP8, autopep8
  - Black
  - YAPF
- Used widely in various areas and communities
  - Research, Software Development, System Development, Scientific Computing, etc.
- References
  - https://peps.python.org/pep-0008/
  - https://pypi.org/project/autopep8/
  - https://github.com/psf/black
  - https://black.readthedocs.io/en/stable/
  - https://pypi.org/project/yapf/0.3.1/
  - https://github.com/google/yapf

# Example 1 - PEP8/autopep8

```
import math, sys;

def example1():
    ####This is a long comment. This should be wrapped to fit within 72 characters.
    some_tuple=(   1,2, 3,'a'  );
    some_variable={'long':'Long code lines should be wrapped within 79 characters.',
    'other':[math.pi, 100,200,300,9876543210,'This is a long string that goes on'],
    'more':{'inner':'This whole logical line should be wrapped.',some_tuple:[1,
    20,300,40000,500000000,60000000000000000]}}
    return (some_tuple, some_variable)
def example2(): return {'has_key() is deprecated':True}.has_key({'f':2}.has_key(''));
class Example3(   object ):
    def __init__    ( self, bar ):
     #Comments should have a space after the hash.
     if bar : bar+=1;  bar=bar* bar   ; return bar
     else:
                    some_string = """
                       Indentation in multiline strings should not be touched.
Only actual code should be reindented.
"""
                    return (sys.path, some_string)
```

```
import math
import sys


def example1():
    # This is a long comment. This should be wrapped to fit within 72
    # characters.
    some_tuple = (1, 2, 3, 'a')
    some_variable = {
        'long': 'Long code lines should be wrapped within 79 characters.',
        'other': [
            math.pi,
            100,
            200,
            300,
            9876543210,
            'This is a long string that goes on'],
        'more': {
            'inner': 'This whole logical line should be wrapped.',
            some_tuple: [
                1,
                20,
                300,
                40000,
                500000000,
                60000000000000000]}}
    return (some_tuple, some_variable)


def example2(): return ('' in {'f': 2}) in {'has_key() is deprecated': True}


class Example3(object):
    def __init__(self, bar):
        # Comments should have a space after the hash.
        if bar:
            bar += 1
            bar = bar * bar
            return bar
        else:
            some_string = """
                       Indentation in multiline strings should not be touched.
Only actual code should be reindented.
"""
            return (sys.path, some_string)
```

# Example 2 - BLACK

```
# in:

j = [1,
     2,
     3
]

# out:

j = [1, 2, 3]
```

```
# in:

def function(
    some_argument: int,

    other_argument: int = 5,
) -> EmptyLineInParenWillBeDeleted:



    print("One empty line above me will be kept!")

def this_is_okay_too():
    print("No empty line here")
# out:

def function(
    some_argument: int,
    other_argument: int = 5,
) -> EmptyLineInParenWillBeDeleted:

    print("One empty line above me will be kept!")


def this_is_okay_too():
    print("No empty line here")
```

```
# in:

ImportantClass.important_method(exc, limit, lookup_lines, capture_locals, extra_argument)

# out:

ImportantClass.important_method(
    exc, limit, lookup_lines, capture_locals, extra_argument
)
```

```
# in:

def very_important_function(template: str, *variables, file: os.PathLike, engine: str, header: boo
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...

# out:

def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

```
# in:

if some_short_rule1 \
  and some_short_rule2:
      ...

# out:

if some_short_rule1 and some_short_rule2:
  ...



# in:

if some_long_rule1 \
  and some_long_rule2:
    ...

# out:

if (
    some_long_rule1
    and some_long_rule2
):
    ...
```

# Example 3 - BLACK

```
# in:

def very_important_function(template: str, *variables, file: os.PathLike, engine: str, header: bo
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...

# out:

def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

# Example 4 - YAPF

```
>>> from yapf.yapf_api import FormatCode  # reformat a string of code

>>> FormatCode("f ( a = 1, b = 2 )")
'f(a=1, b=2)\n'
```

```
>>> FormatCode("def g():\n  return True", style_config='pep8')
'def g():\n    return True\n'
```

```
>>> FormatCode("def g( ):\n    a=1\n    b = 2\n    return a==b", lines=[(1, 1), (2, 3)])
'def g():\n    a = 1\n    b = 2\n    return a==b\n'
```

```
>>> print(FormatCode("a==b", filename="foo.py", print_diff=True))
--- foo.py (original)
+++ foo.py (reformatted)
@@ -1 +1 @@
-a==b
+a == b
```

# PEP8 - Python Enhancement Proposal-8

- **Official Documentation** - https://peps.python.org/pep-0008/
- Feature-Rich
  - Code Lay-out: Indentation | Tab or Spaces? | Maximum Line Length | Blank Lines | Source File Encoding | Imports | Module Level Dunder Names
  - String Quotes
  - Comments: Block Comments | Inline Comments | Documentation Strings
  - Naming Conventions: Overriding Principle | Naming Styles | Naming Conventions
    - Names to Avoid | ASCII Compatibility | Package and Module Names | Class Names | Type Variable Names | Exception Names | Global Variable Names | Function and Variable Names | Function and Method Arguments | Method Names and Instance Variables | Constants | Designing for Inheritance
  - Programming Recommendations
    - Functional Annotations
    - Variable Annotations

# PEP8 & autopep8

- In general, some preprocessing on the existing code is first performed (parsing, AST producing, extra code presentations, etc)
    - Then, some tree traversal and editing is perform
    - Example:

```
import ast
code = """
def hello_world():
        print("CS 846 - ATEP") """
tree = ast.parse(code)
print(ast.dump(tree))
```

- Common
    - Indentation
    - Redundant black lines
    - Docstrings
    - Number of line breaks in each region
    - Comments
- It’s beneficial to complete all this, but better to delegate to some automation tools

# BLACK

- **Official Documentation** - https://black.readthedocs.io/en/stable/the_black_code_style/index.html
- Condensed Features
  - Wrap lines | Line length | Empty lines | Comments | Strings | Commas | Slices | Parentheses | Call chains
- Additional
  - Black also checks whether the produced code and the previous code is semantically equivalent
  - AST based algorithm is a little more robust, capable of cleaning up different kinds of whitespaces
  - Capable of manage different kinds of parentheses and set corresponding configurations
  - Powerful string processing algorithms
  - Like other tools, but Black seems to integrate with a lot of more tools
    - Wing IDE, Vim, With ALE, Gedit, Visual Studio Code, SublimeText, Python LSP Server, Atom/Nuclide, Gradle, Kakoune, Thonny
    - Even GitHub actions integration & Version control integration
- Straightforward to add to the project

# BLACK

- Straightforward and streamlined development process (easier to maintain and fix bugs, one of the core reason making it the most popular out of the 3)
- Resources
  - https://black.readthedocs.io/en/stable/contributing/the_basics.html
  - https://black.readthedocs.io/en/stable/contributing/gauging_changes.html
  - https://black.readthedocs.io/en/stable/contributing/issue_triage.html
  - https://black.readthedocs.io/en/stable/contributing/release_process.html

## Technicalities

Development on the latest version of Python is preferred. You can use any operating system.

Install development dependencies inside a virtual environment of your choice, for example:

```
$ python3 -m venv .venv
$ source .venv/bin/activate # activation for linux and mac
$ .venv\Scripts\activate # activation for windows

(.venv)$ pip install -r test_requirements.txt
(.venv)$ pip install -e .[d]
(.venv)$ pre-commit install
```

Before submitting pull requests, run lints and tests with the following commands from the root of the black repo:

```
# Linting
(.venv)$ pre-commit run -a

# Unit tests
(.venv)$ tox -e py

# Optional Fuzz testing
(.venv)$ tox -e fuzz

# Format Black itself
(.venv)$ tox -e run_self
```

## Development

Further examples of invoking the tests

```
# Run all of the above mentioned, in parallel
(.venv)$ tox --parallel=auto

# Run tests on a specific python version
(.venv)$ tox -e py39

# pass arguments to pytest
(.venv)$ tox -e py -- --no-cov

# print full tree diff, see documentation below
(.venv)$ tox -e py -- --print-full-tree

# disable diff printing, see documentation below
(.venv)$ tox -e py -- --print-tree-diff=False
```

# YAPF - Yet Another Python Formatter

- **Official Documentation** - https://github.com/google/yapf
- Condensed Features (& Highly Configurable)
    - Wrapper of various styles
    - Can switch between the styles
    - Can set extra configurations
    - Complex configurations
    - Examples:
        - `BLANK_LINES_BETWEEN_TOP_LEVEL_IMPORTS_AND_VARIABLES`
        - `CONTINUATION_ALIGN_STYLE`
        - `EACH_DICT_ENTRY_ON_SEPARATE_LINE`
        - `INDENT_CLOSING_BRACKETS`
        - `INDENT_DICTIONARY_VALUE`
        - `SPACE_BETWEEN_ENDING_COMMA_AND_CLOSING_BRACKET`
        - `SPLIT_BEFORE_EXPRESSION_AFTER_OPENING_PAREN`
        - `SPLIT_PENALTY_LOGICAL_OPERATOR`

# YAPF - Yet Another Python Formatter

- Based on clang-format
- High-configurable
- “The algorithm takes the code and calculates the best formatting that conforms to the configured style. It takes away a lot of the drudgery of maintaining your code.”
- “YAPF's formatting algorithm creates a weighted tree that acts as the solution space for the algorithm. Each node in the tree represents the result of a formatting decision --- i.e., whether to split or not to split before a token. Each formatting decision has a cost associated with it. Therefore, the cost is realized on the edge between two nodes. (In reality, the weighted tree doesn't have separate edge objects, so the cost resides on the nodes themselves.)”

# Use Cases

- There are many use cases. Let’s discuss some common use cases
    - Some are available in one tools but not the others (but there can be overlaps between feature design and implementation)
    - Overall, the general idea is to present the existing the code to some data structure and run some algorithms on there
- Use cases
    - Removing redundant whitespaces
    - Enforce function naming conventions
    - Code regions
    - Enforce line length (line wrapping)
    - Converting statements
    - Optimize imports
    - Redundant blank lines
    - etc.
    - ***Highly customizable!***

# Use Case 1 - Removing redundant whitespaces

```python
from seven_dwwarfs import Grumpy, Happy, Sleepy, Bashful, Sneezy, Dopey, Doc
x = {  'a':37,'b':42,

'c':927}

x = 123456789.123456789E123456789

if very_long_variable_name is not None and \
 very_long_variable_name.field > 0 or \
 very_long_variable_name.is_debug:
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (this
and that): y = 'hello ''world'#FIXME: https://github.com/psf/black/issues/26
class Foo  (     object  ):
  def f    (self   ):
    return       37*-2





  def g(self, x,y=42):
      return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
```

```python
from seven_dwwarfs import Grumpy, Happy, Sleepy, Bashful, Sneezy, Dopey, Doc

x = {"a": 37, "b": 42, "c": 927}

x = 123456789.123456789e123456789

if (
    very_long_variable_name is not None
    and very_long_variable_name.field > 0
    or very_long_variable_name.is_debug
):
    z = "hello " + "world"
else:
    world = "world"
    a = "hello {}".format(world)
    f = rf"hello {world}"
if this and that:
    y = "hello " "world"  # FIXME: https://github.com/psf/black/issues/26


class Foo(object):
    def f(self):
        return 37 * -2

    def g(self, x, y=42):
        return y


def f(a: List[int]):
    return 37 - a[42 - u : y**3]


def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

# Use Case 2 - Enforcing maximum line-length

```python
    def g(self, x,y=42):
        return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):  # placehc
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
```

```python
23          return 37 * -2
24
25      def g(self, x, y=42):
26          return y
27
28
29  def f(a: List[int]):
30      return 37 - a[42 - u : y**3]
31
32
33  def very_important_function(
34      template: str,
35      *variables,
36      file: os.PathLike,
37      debug: bool = False,
38  ):  # placeholder for a very long code region
39      """Applies `variables` to the `template` and writes to `file`."""
40      with open(file, "w") as f:
41          ...
42
```

# Use Case 3 - Enforcing quotation marks

```
if very_long_variable_name is not None and \
 very_long_variable_name.field > 0 or \
 very_long_variable_name.is_debug:
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
```

```
7   if (
8       very_long_variable_name is not None
9       and very_long_variable_name.field > 0
10      or very_long_variable_name.is_debug
11  ):
12      z = "hello " + "world"
13  else:
14      world = "world"
```

```
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    '''Applies `variables` to the `template` and writes to `file`.'''
    with open(file, "w") as f:
     ...
# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
# fmt: on
regular_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
```

```
24
25      def g(self, x, y=42):
26          return y
27
28
29  def f(a: List[int]):
30      return 37 - a[42 - u : y**3]
31
32
33  def very_important_function(
34      template: str,
35      *variables,
36      file: os.PathLike,
37      debug: bool = False,
38  ):
39      """Applies `variables` to the `template` and writes to `file`."""
40      with open(file, "w") as f:
41          ...
42
43
44  # fmt: off
45  custom_formatting = [
46      0,  1,  2,
47      3,  4,  5,
```

# Use Case 4 - White spaces between code regions

```python
x = {  'a':37,'b':42,

'c':927}

x = 123456789.123456789E123456789

if very_long_variable_name is not None and \
 very_long_variable_name.field > 0 or \
 very_long_variable_name.is_debug:
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (this
and that): y = 'hello ''world'#FIXME: https://github.com/psf/black/issues/26
```

```python
class Foo  (     object  ):
  def f    (self   ):
    return       37*-2
  def g(self, x,y=42):
      return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
# fmt: on
regular_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
```

```python
2
3  x = {"a": 37, "b": 42, "c": 927}
4
5  x = 123456789.123456789e123456789
6
7  if (
8      very_long_variable_name is not None
9      and very_long_variable_name.field > 0
10     or very_long_variable_name.is_debug
11 ):
12     z = "hello " + "world"
13 else:
14     world = "world"
15     a = "hello {}".format(world)
16     f = rf"hello {world}"
17 if this and that:
18     y = "hello " "world"  # FIXME: https://github.com/psf/black/issues/26
19
20
21 class Foo(object):
22     def f(self):
23         return 37 * -2
24
25     def g(self, x, y=42):
26         return y
27
28
29 def f(a: List[int]):
30     return 37 - a[42 - u : y**3]
31
32
33 def very_important_function(
34     template: str,
35     *variables,
36     file: os.PathLike,
37     debug: bool = False,
38 ):
39     """Applies `variables` to the `template` and writes to `file`."""
40     with open(file, "w") as f:
41         ...
42
43
44 # fmt: off
45 custom_formatting = [
46     0,  1,  2,
47     3,  4,  5,
48     6,  7,  8,
49 ]
50 # fmt: on
51 regular_formatting = [
52     0,
53     1,
54     2,
55     3,
56     4,
57     5,
58     6,
59     7,
60     8,
61 ]
62
```

# Use Case 5 - White spaces between lines

```
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (this
and that): y = 'hello ''world'#FIXME: https://github.com/psf/black/issues/26



class Foo  (     object  ):
  def f    (self   ):



    return      37*-2
  def g(self, x,y=42):
      return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
# fmt: on
regular_formatting = [
```

```
else:
    world = "world"
    a = "hello {}".format(world)
    f = rf"hello {world}"
if this and that:
    y = "hello " "world"  # FIXME: https://github.


class Foo(object):
    def f(self):

        return 37 * -2

    def g(self, x, y=42):
        return y


def f(a: List[int]):
    return 37 - a[42 - u : y**3]


def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,
):
    """Applies `variables` to the `template` and w
    with open(file, "w") as f:
        ...


# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
# fmt: on
regular_formatting = [
    0,
    1,
    2,
    3,
    4,
    5,
    6,
    7,
```

# Use Case 6 - Reindent Code

```
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (this
and that): y = 'hello ''world'#FIXME: https://github.com/psf/black/issues/26
class Foo  (     object  ):
  def f    (self   ):
    return       37*-2
  def g(self, x,y=42):
      return y
def f  (   a: List[ int ]) :
  return      37-a[42-u :  y**3]
def very_important_function(template: str,*variables,file: os.PathLike,debug:bool=False,):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
     ...
# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
# fmt: on
regular_formatting = [
    0,  1,  2,
    3,  4,  5,
    6,  7,  8,
]
```

```
13 else:
14     world = "world"
15     a = "hello {}".format(world)
16     f = rf"hello {world}"
17 if this and that:
18     y = "hello " "world"  # FIXME: https:/
19
20
21 class Foo(object):
22     def f(self):
23         return 37 * -2
24
25     def g(self, x, y=42):
26         return y
27
28
29 def f(a: List[int]):
30     return 37 - a[42 - u : y**3]
31
32
33 def very_important_function(
34     template: str,
35     *variables,
36     file: os.PathLike,
37     debug: bool = False,
38 ):
39     """Applies `variables` to the `template
40     with open(file, "w") as f:
41         ...
42
43
44 # fmt: off
45 custom_formatting = [
46     0,  1,  2,
47     3,  4,  5,
48     6,  7,  8,
```

# Summary

- AST based tools are powerful in code formatting
  - Efficient
  - Easy to add features
- Ongoing research areas
  - Better styles
  - More efficient algorithms
  - Less memory usage
- Code formatting is an important task in software engineering
  - Many benefits
  - Identify potential errors
  - Can potentially help saving some costs
- Many tools for code formatting
  - Different tools, but there can be common functionalities
  - Should be easy to add features to the tool