Technische
Universität
Braunschweig

Machine learning

Mini-project: PEP 8

Kristian Siebenrock

Table of Contents

1. What is PEP 8?

2. Why *should* it be used?

3. Key Points

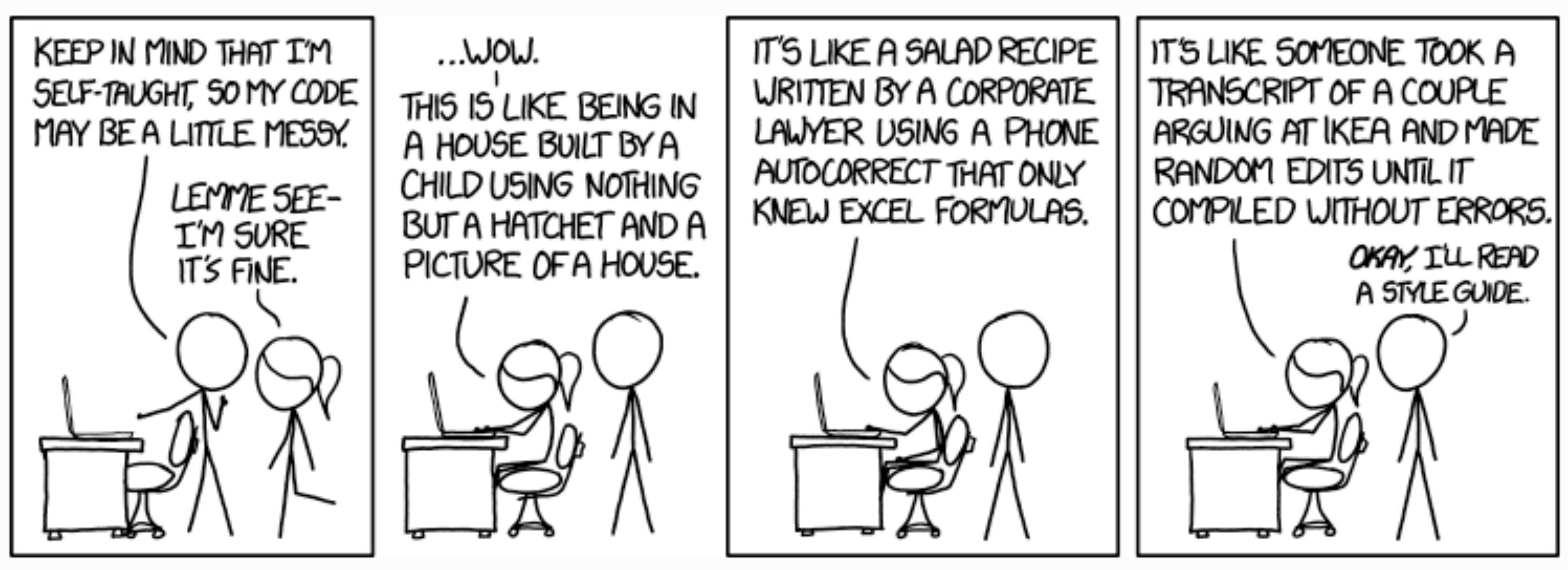
4. When *shouldn’t* it be used?

5. Implementation

Table of Contents

1. **What is PEP 8?**
2. Why *should* it be used?
3. Key Points
4. When *shouldn't* it be used?
5. Implementation

What is PEP 8?


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

https://geo-python.github.io/site/notebooks/L3/gcp-3-pep8.html

What is PEP 8?

- PEP (Python Enhancement Proposal): A document that describes and documents new features for the community
- PEP 8 is a document containing guidelines and best practices for writing Python code.
- Focus: Improve readability and consistency of Python code
- Written by Guido van Rossum, Barry Warsaw and Nick Coghlan in 2001

Table of Contents

1. What is PEP 8?
2. **Why *should* it be used?**
3. Key Points
4. When *shouldn’t* it be used?
5. Implementation

Why *should* it be used?

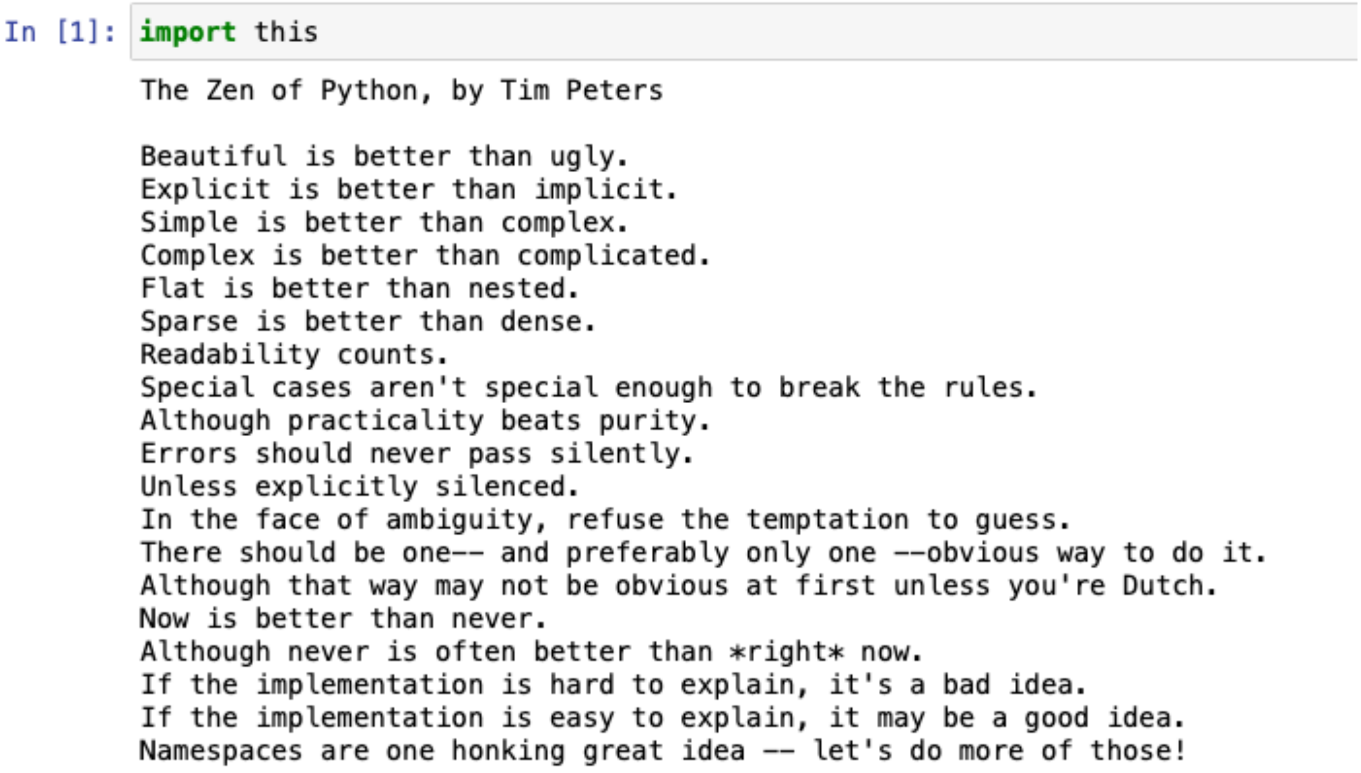
“Code is read much more often than it is written.”

- Guido van Rossum

Why *should* it be used?

“Code is read much more often than it is written.”

- Guido van Rossum



```
In [1]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Why *should* it be used?

Improve readability and consistency

Shows professionalism

Improved experience while collaborating with others

Table of Contents

1. What is PEP 8?

2. Why *should* it be used?

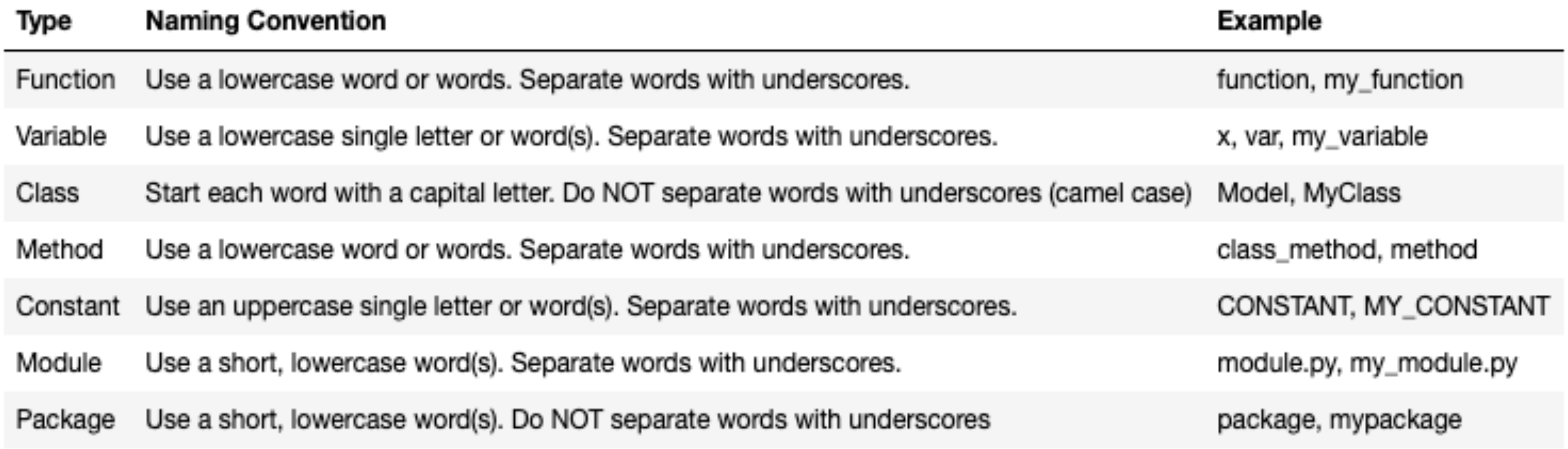
3. **Key Points**
    1. Naming Conventions
    2. Code Layout
    3. Indentation
    4. Comments
    5. Whitespace

4. When *shouldn't* it be used?

5. Implementation

Key Points – Naming Conventions

“Explicit is better than Implicit”
- The Zen of Python

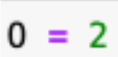


Type	Naming Convention	Example
Function	Use a lowercase word or words. Separate words with underscores.	function, my_function
Variable	Use a lowercase single letter or word(s). Separate words with underscores.	x, var, my_variable
Class	Start each word with a capital letter. Do NOT separate words with underscores (camel case)	Model, MyClass
Method	Use a lowercase word or words. Separate words with underscores.	class_method, method
Constant	Use an uppercase single letter or word(s). Separate words with underscores.	CONSTANT, MY_CONSTANT
Module	Use a short, lowercase word(s). Separate words with underscores.	module.py, my_module.py
Package	Use a short, lowercase word(s). Do NOT separate words with underscores	package, mypackage

Key Points – Naming Conventions

Choosing Names:

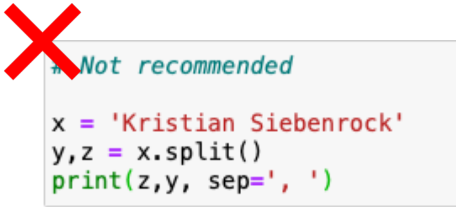
1. Never use l, O or I single letter names:



```
O = 2
```

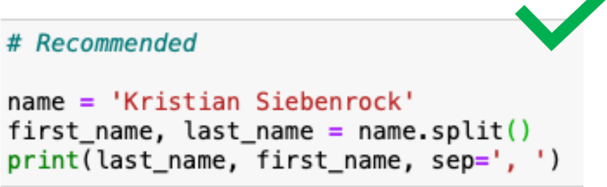
2. Do not use x, y or z when naming varibales:

✗



```
# Not recommended

x = 'Kristian Siebenrock'
y,z = x.split()
print(z,y, sep=', ')
```

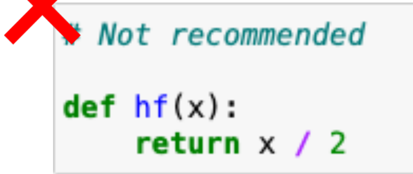
✓



```
# Recommended

name = 'Kristian Siebenrock'
first_name, last_name = name.split()
print(last_name, first_name, sep=', ')
```

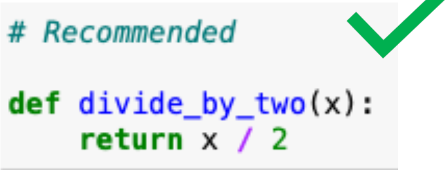
3. Try not to use abbreviations:

✗



```
# Not recommended

def hf(x):
    return x / 2
```

✓



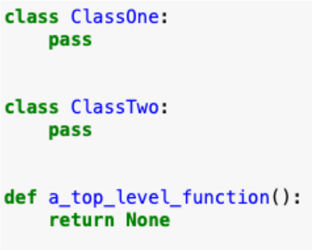
```
# Recommended

def divide_by_two(x):
    return x / 2
```

Key Points – Code Layout

“Beautiful is better than ugly”
- The Zen of Python

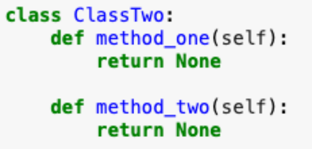
Blank lines:

1. Surround top-level functions and classes with two blank lines:



```python
class ClassOne:
    pass


class ClassTwo:
    pass


def a_top_level_function():
    return None
```

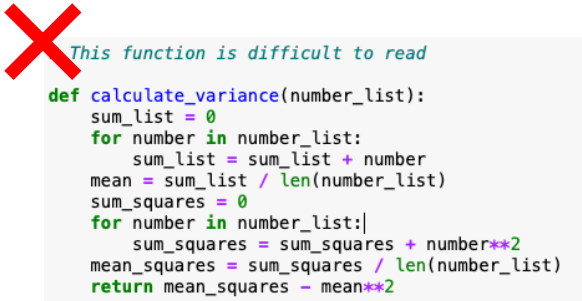
2. Surround method definitions inside classes with a single blank line:



```python
class ClassTwo:
    def method_one(self):
        return None

    def method_two(self):
        return None
```

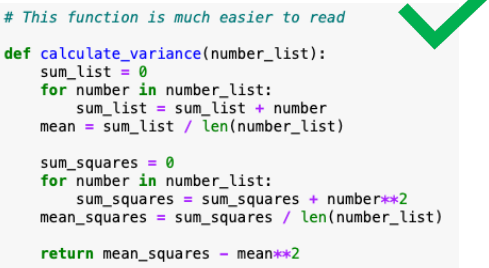
3. Use blank lines sparingly inside functions to depict steps:

✗



```python
# This function is difficult to read

def calculate_variance(number_list):
    sum_list = 0
    for number in number_list:
        sum_list = sum_list + number
    mean = sum_list / len(number_list)
    sum_squares = 0
    for number in number_list:
        sum_squares = sum_squares + number**2
    mean_squares = sum_squares / len(number_list)
    return mean_squares - mean**2
```

✓



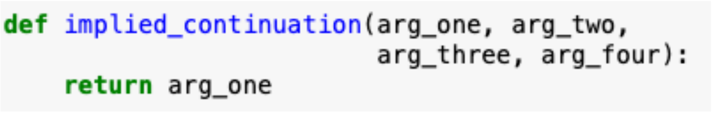
```python
# This function is much easier to read

def calculate_variance(number_list):
    sum_list = 0
    for number in number_list:
        sum_list = sum_list + number
    mean = sum_list / len(number_list)

    sum_squares = 0
    for number in number_list:
        sum_squares = sum_squares + number**2
    mean_squares = sum_squares / len(number_list)

    return mean_squares - mean**2
```

Key Points – Code Layout

Maximum Line Length: PEP 8 suggests that lines should be limited to 79 characters

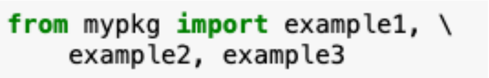
Line Breaking:

- Implied Continuation



```python
def implied_continuation(arg_one, arg_two,
                         arg_three, arg_four):
    return arg_one
```

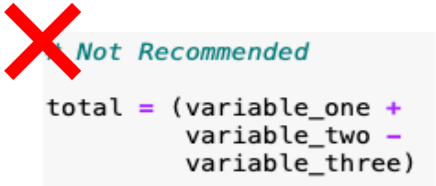
- Use Backslashes to break lines:



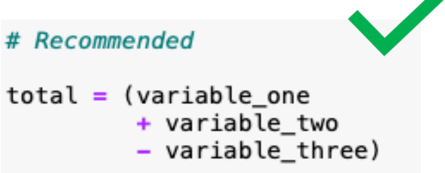
```python
from mypkg import example1, \
    example2, example3
```

- Break lines before binary operators:

✗



```python
# Not Recommended

total = (variable_one +
         variable_two -
         variable_three)
```



```python
# Recommended

total = (variable_one
         + variable_two
         - variable_three)
```

✓

Key Points – Indentation

“There should be one—and preferably only one—obvious way to do it.”
- The Zen of Python

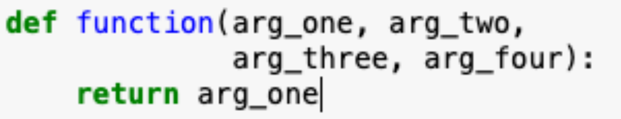
1. Prefer spaces over tabs

2. Use 4 consecutive spaces to indicate indentation

Key Points – Indentation

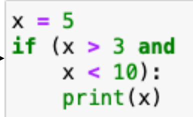
Indentation following line breaks:

1. Align the indented block with the opening delimeter

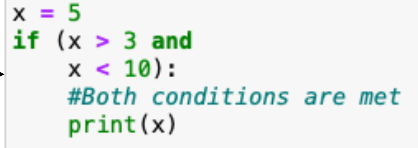


```python
def function(arg_one, arg_two,
             arg_three, arg_four):
    return arg_one
```

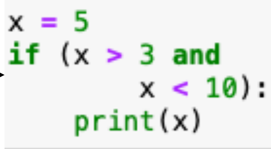
If



```python
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

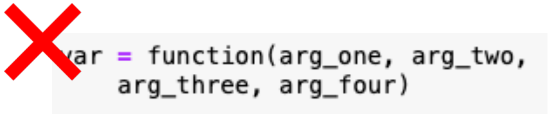


```python
x = 5
if (x > 3 and
    x < 10):
    #Both conditions are met
    print(x)
```

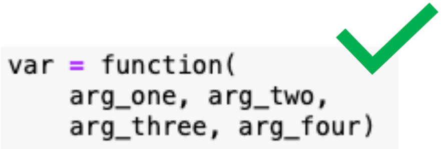


```python
x = 5
if (x > 3 and
        x < 10):
    print(x)
```

2. Hanging Indent

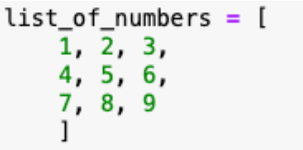


```python
var = function(arg_one, arg_two,
    arg_three, arg_four)
```

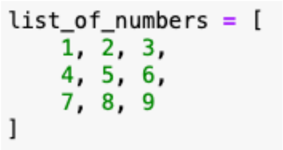


```python
var = function(
    arg_one, arg_two,
    arg_three, arg_four)
```

3. Breaking lines inside parenthesis, brackets or braces:



```python
list_of_numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
    ]
```



```python
list_of_numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]
```

Key Points – Comments

“If the implementation is hard to explain, it’s a bad idea.”
- The Zen of Python

1. Limit the line length of comments and docstrings to 72 characters

2. Use complete sentences, starting with a capital letter

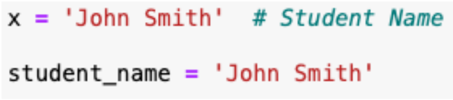
3. Make sure to update comments if the code is changed

! Comments should generally always be written in English !

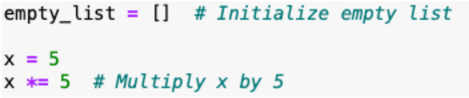
Key Points – Comments

- *Inline Comments*

  1. Use inline comments sparingly
  2. Write inline comments on the same line as the statement they refer to
  3. Separate inline comments by two or more spaces from the statement
  4. Start inline comments with a # and a single space
  5. Do not use them to state the obvious



```
x = 'John Smith'  # Student Name

student_name = 'John Smith'
```



```
empty_list = []  # Initialize empty list

x = 5
x *= 5  # Multiply x by 5
```

- *Block Comments*

  1. Indent block comments to the same level as the code they describe
  2. Start each line with a # and a single space
  3. Separate paragraphs by a line containing a single #

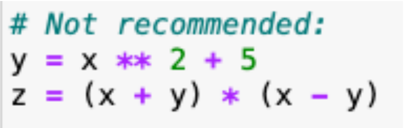
- *Documentation Strings*

  1. Surround docstrings with three double quotes on either side
  2. Write them for all public modules, functions, classes and methods
  3. Put the “ “ ” that ends a multiple docstring on a line by itself

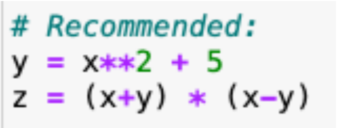
Key Points – Whitespace

“Sparse is better than dense”
- The Zen of Python

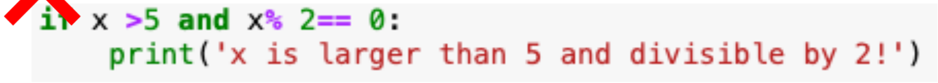
- Whitespace should be surrounded by the following operators:
  - Assignment Operators (=, +=, -=, …)
  - Comparisons (==, !=, >, <, …)
  - Booleans (and, or, not)
- Only add whitespace to operators with the lowest priority:



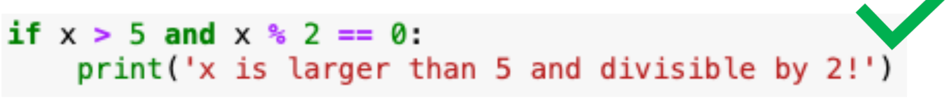
```
# Not recommended:
y = x ** 2 + 5
z = (x + y) * (x - y)
```



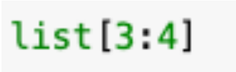
```
# Recommended:
y = x**2 + 5
z = (x+y) * (x-y)
```



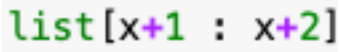
```
if x >5 and x% 2== 0:
    print('x is larger than 5 and divisible by 2!')
```



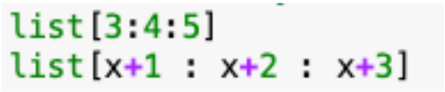
```
if x > 5 and x % 2 == 0:
    print('x is larger than 5 and divisible by 2!')
```



```
list[3:4]
```



```
list[x+1 : x+2]
```

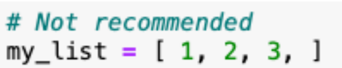


```
list[3:4:5]
list[x+1 : x+2 : x+3]
```

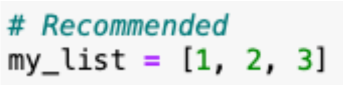
Key Points – Whitespace

When to avoid adding whitespace:

1. Immediately inside parenthesis, brackets or braces

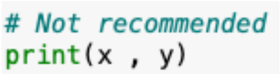


```
# Not recommended
my_list = [ 1, 2, 3, ]
```

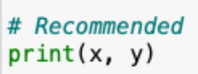


```
# Recommended
my_list = [1, 2, 3]
```

2. Before a comma, semicolon or colon

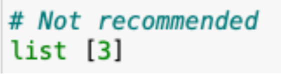


```
# Not recommended
print(x , y)
```

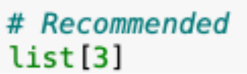


```
# Recommended
print(x, y)
```

3. Before the open bracket that starts an index or slice

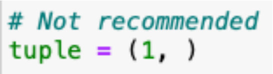


```
# Not recommended
list [3]
```

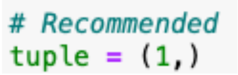


```
# Recommended
list[3]
```

4. Between a trailing comma and a closing parenthesis

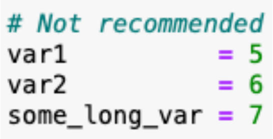


```
# Not recommended
tuple = (1, )
```

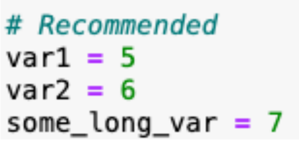


```
# Recommended
tuple = (1,)
```

5. To align assignment operators



```
# Not recommended
var1          = 5
var2          = 6
some_long_var = 7
```



```
# Recommended
var1 = 5
var2 = 6
some_long_var = 7
```

Table of Contents

1. What is PEP 8?
2. Why *should* it be used?
3. Key Points
4. **When *shouldn’t* it be used?**
5. Implementation

Why *shouldn’t* it be used?

If complying would result in breaking backwards compatibility

When applying a guidline would make the code less readable

To be consistent with code that does not adhere to it

When code needs to be compatible with older versions of Python that don’t support certain features

Table of Contents

1. What is PEP 8?
2. Why *should* it be used?
3. Key Points
4. When *shouldn't* it be used?
5. **Implementation**
    1. **Linters**
    2. **Autoformatters**

Implementation – Linters

Linters are programs that analyse code and flag errors and provide suggestions on how to fix the error in regard to PEP 8.

⊕ Especially useful when installed as extensions to a text editor, as they flag errors and stylistic problems in real time

⊖ Only report the problems they identify in the source code and leave the changing of the code to the developers

Most popular linters for Python:

- pycodestyle: https://pycodestyle.pycqa.org/en/latest/
- Pylint: https://www.pylint.org
- Flake8: https://flake8.pycqa.org/en/latest/

Implementation – Linters

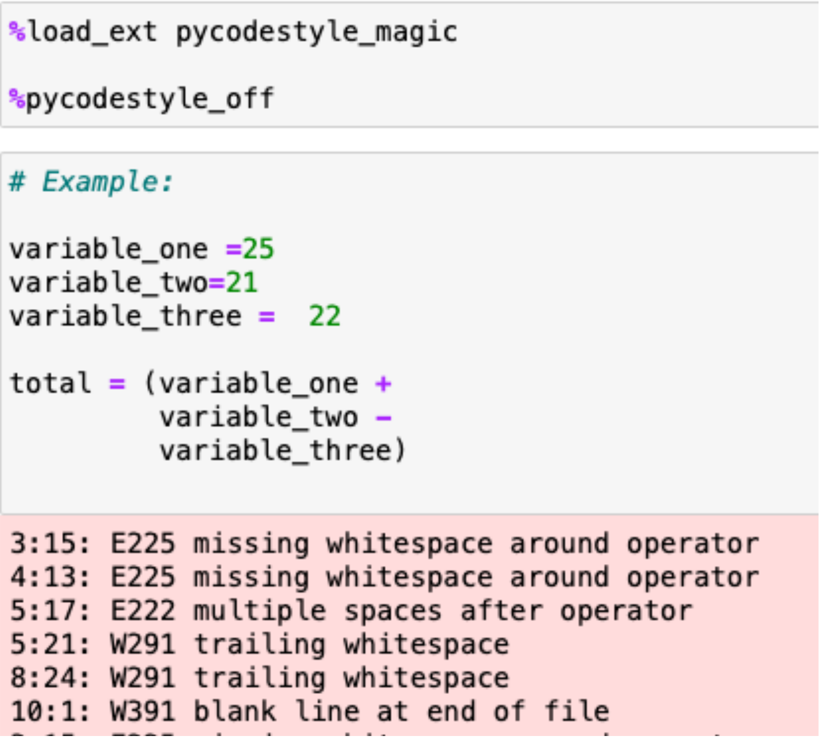
1. pycodestyle

Features that are able to be checked:

- Indentation
- Whitespace
- Blank lines
- Import
- Line length
- Runtime
- Line Breaks

Features not in the scope:

- Naming conventions
- Docstring conventions



```
%load_ext pycodestyle_magic

%pycodestyle_off
```

```
# Example:

variable_one =25
variable_two=21
variable_three =  22

total = (variable_one +
         variable_two -
         variable_three)
```

```
3:15: E225 missing whitespace around operator
4:13: E225 missing whitespace around operator
5:17: E222 multiple spaces after operator
5:21: W291 trailing whitespace
8:24: W291 trailing whitespace
10:1: W391 blank line at end of file
```

Implementation – Linters

Implementing pycodestyle in Jupyter Notebook:

Step 1: pip install pycodestyle

Step 2: pip install pycodestyle_magic

Step 3: %load_ext pycodestyle_magic

Step 4: %pycodestyle_on

Implementation – Linters

2. Pylint

Pylint

Star your Python code!

https://www.pylint.org

- Static code checker, unlike pycodestyle
- Most commonly used tool for linting in Python

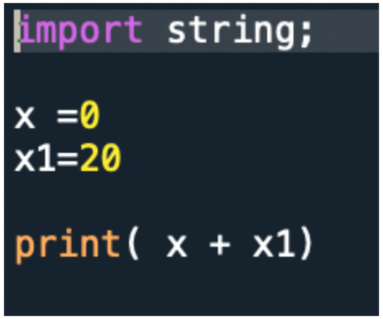
⊕

- Has more error/warning checks than many other linters
- More descriptive
- Delivers a code rating and compares to previous versions
- Integrated in numerous editors

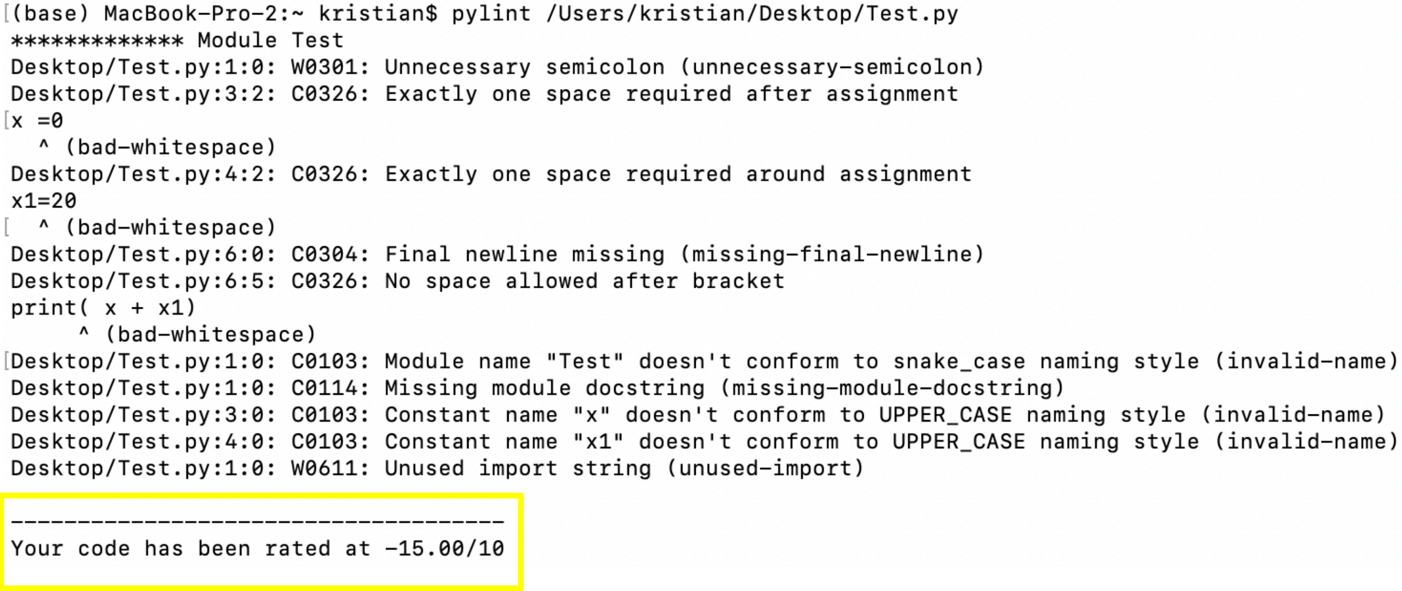
⊖

- Cannot be implemented in Jupyter Notebook
- Can only take .py files

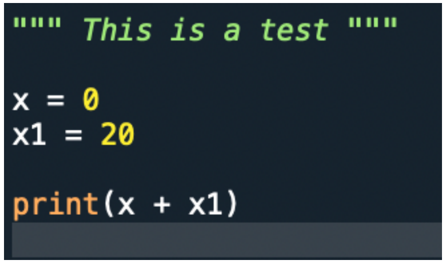
Implementation – Linters



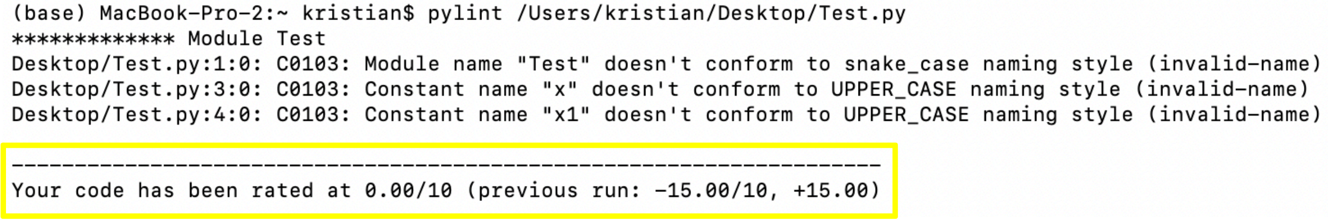
```
import string;

x =0
x1=20

print( x + x1)
```



```
(base) MacBook-Pro-2:~ kristian$ pylint /Users/kristian/Desktop/Test.py
************* Module Test
Desktop/Test.py:1:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
Desktop/Test.py:3:2: C0326: Exactly one space required after assignment
x =0
  ^ (bad-whitespace)
Desktop/Test.py:4:2: C0326: Exactly one space required around assignment
x1=20
  ^ (bad-whitespace)
Desktop/Test.py:6:0: C0304: Final newline missing (missing-final-newline)
Desktop/Test.py:6:5: C0326: No space allowed after bracket
print( x + x1)
     ^ (bad-whitespace)
Desktop/Test.py:1:0: C0103: Module name "Test" doesn't conform to snake_case naming style (invalid-name)
Desktop/Test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
Desktop/Test.py:3:0: C0103: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid-name)
Desktop/Test.py:4:0: C0103: Constant name "x1" doesn't conform to UPPER_CASE naming style (invalid-name)
Desktop/Test.py:1:0: W0611: Unused import string (unused-import)

-------------------------------------
Your code has been rated at -15.00/10
```



```
""" This is a test """

x = 0
x1 = 20

print(x + x1)
```



```
(base) MacBook-Pro-2:~ kristian$ pylint /Users/kristian/Desktop/Test.py
************* Module Test
Desktop/Test.py:1:0: C0103: Module name "Test" doesn't conform to snake_case naming style (invalid-name)
Desktop/Test.py:3:0: C0103: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid-name)
Desktop/Test.py:4:0: C0103: Constant name "x1" doesn't conform to UPPER_CASE naming style (invalid-name)

-------------------------------------------------------------------
Your code has been rated at 0.00/10 (previous run: -15.00/10, +15.00)
```

Implementation – Linters

Implementing Pylint:

Step 1: pip install pylint

Step 2: pip module1.py, (module2.py,…)

Using Pylint in other editors:

Visual Studio:	Python > Run Pylint
Spyder:	View > Panes > Static code analysis

Implementation – Auto-Formatters

Auto-formatters are tools that will format code in a way that complies with PEP 8.

⊕

- Fixes inconsistencies instead of just raising warnings/errors
- Uniform style after auto-formatting

⊖

- Removes flexibility in regard to formatting

Most popular Auto-formatters for Python:

- Autopep8: https://github.com/hhatto/autopep8#features
- Black: https://github.com/psf/black

Implementation – Auto-Formatters

1. Black

https://github.com/psf/black#the-black-code-style

- One of the most popular auto-formatters for PEP 8 compliance
- Reformats entire files in place

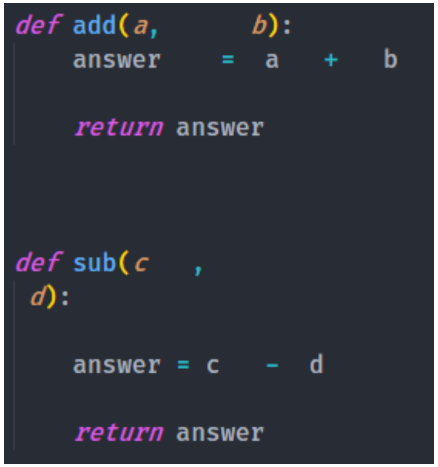
⊕

- Fast
- Transparent
- Blocks of code can be selected to not be formatted
- Can be integrated into numerous editors

⊖

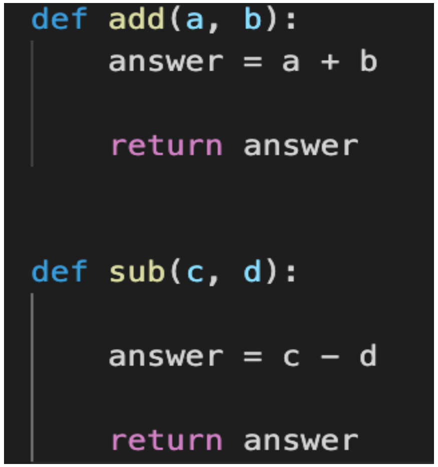
- Not configurable
- Doesn’t take previous formatting into account

Implementation – Auto-Formatters



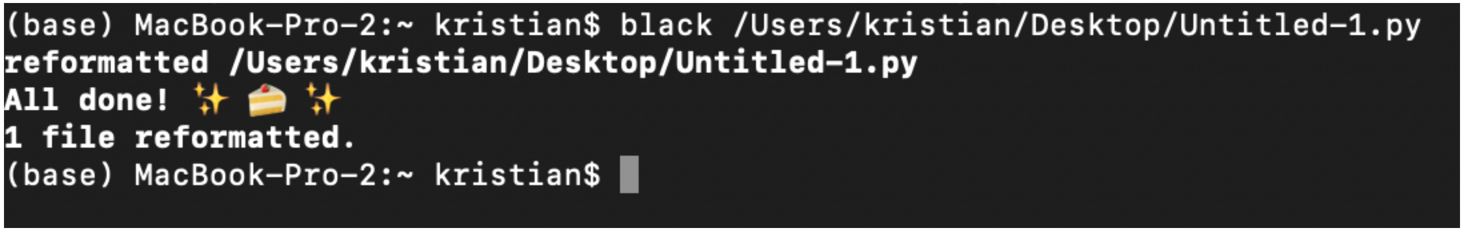
```
def add(a,       b):
    answer     =  a    +    b

    return answer



def sub(c    ,
 d):

    answer = c    -  d

    return answer
```

✗



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

    return answer


def sub(c, d):

    answer = c - d

    return answer
```

✓

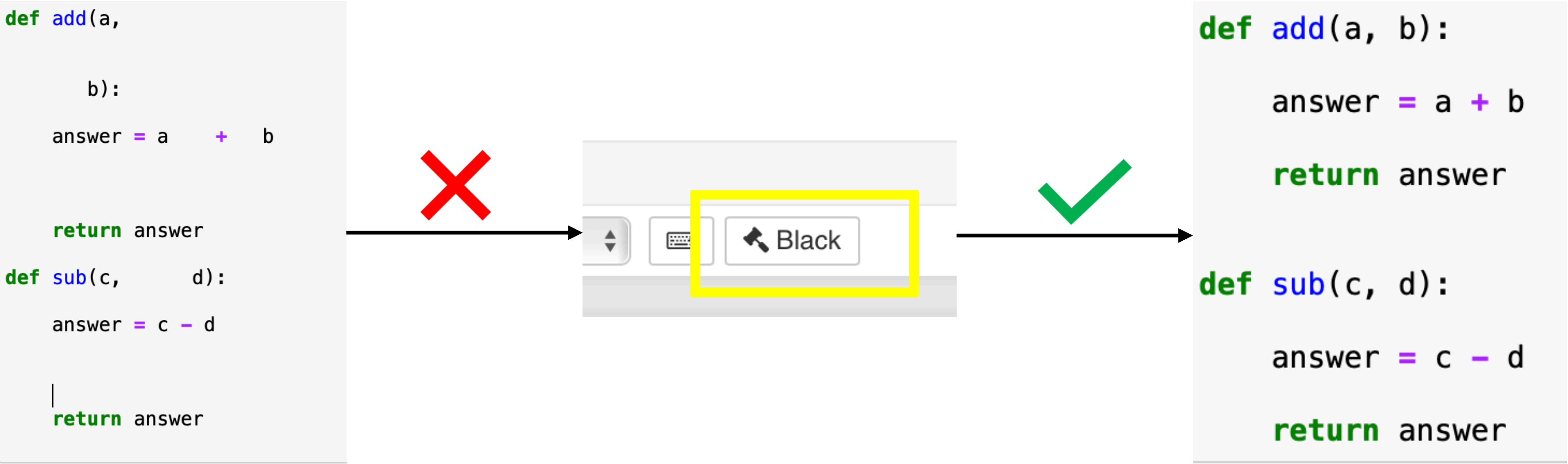


```
(base) MacBook-Pro-2:~ kristian$ black /Users/kristian/Desktop/Untitled-1.py
reformatted /Users/kristian/Desktop/Untitled-1.py
All done! ✨ 🍰 ✨
1 file reformatted.
(base) MacBook-Pro-2:~ kristian$ 
```

Implementation – Auto-Formatters

1. Jupyter Black

- Jupyter Notebook version of Black
- Jupyter Black reformats code in a notebooks cell.
  - Therefore it is possible to just reformat certain cells



```
def add(a,

        b):
    answer = a     +    b


    return answer
def sub(c,       d):
    answer = c - d


    return answer
```

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


def sub(c, d):
    answer = c - d
    return answer
```

Implementation – Auto-Formatters

Implementing Black:

Step 1: pip install black

Step 2: black module1.py, (module2.py,…)

Implementing Jupyter Black:

Step 1: pip install jupyter_contrib_nbextensions

Step 2: jupyter nbextension install https://github.com/drillan/jupyter-black/archive/master.zip --user

Step 3: jupyter nbextension enable jupyter-black-master/jupyter-black

Step 4: Open notebook and click on 'Black' button on desired cell to apply

References

https://pep8.org

https://en.wikipedia.org/wiki/Zen_of_Python

https://www.codeflow.site/de/article/python-pep8

https://realpython.com/python-pep8/

https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it

https://pypi.org/project/pycodestyle/

http://pylint.pycqa.org/en/latest/intro.html

https://www.freecodecamp.org/news/auto-format-your-python-code-with-black/