Coding Style Guide

Introduction to Python Programming

2022-2023

The following mild coding standard ensures easily readable and modifiable Python source code. This standard is based on [1], where much more detailed rules, motivations, and examples can be found. The notes below explain these rules. Beware that some of the examples lack type hints and docstrings, even though these ones are of mandatory use within the context of the course. We do this to avoid distracting your attention from the style guideline.

[1]

Motivation

Well-organized source code is important for several reasons. For instance,

- The compiler may not care about this, but source code is also read by others (e.g., developers, reviewers, maintainers).
- It is an important means to *prevent defects*.
- It facilitates the *localization of defects*, both by the author and by others.

Guidelines

This section highlights the most important style guidelines used when writing Python code. Additional code examples are presented along the section.

Naming Conventions

The name of modules, classes, methods, functions, and variables should reflect their purpose. Good names help with the readability of the code. In addition to that, you should be aware of the following naming styles when using one construct or the other:

- Class names should *always* start with a capital letter so classes are easily spotted. Any additional words in the class name should also start with a capital letter. This is the CapWords or CamelCase convention (e.g., `KittenShop`).
- Modules, functions, methods, and variables should be written in lowercase with words separated by underscores (e.g., `sum_integers`).

- Constants are written in all caps to remind developers that these values should not and will not be changed. If the constant has multiple words, they must be separated by underscores (e.g., `MAX_VALUE`).

As a special case, use always the word `self` as the first argument of any instance method. Lastly, if a name clashes with a reserved Python keyword, add a trailing underscore instead of abbreviating the name (e.g., `type_` instead of `typ`).

Type Hints

Functions and methods often only allow a specific type of variables to be given as arguments. Because code is often created in a team, other members may not know what variable types are allowed in a function or method. Type hints help developers get to know the accepted argument and return types. The only parameter that doesn't need a type hint is the parameter `self` when defining a method. In this course, we use the `typing` module to denote complex data structures such as `List` and `Tuple`. Return type and parameter type hints are written as follows:

```
from typing import List, Dict

def extract_prices(products: List[Dict]) -> List[int]:
    # Extracts the prices from the list of dictionaries.
```

Indentation

Indentation provides visual clues about the containment structure (*nesting*) of the code. Prefer to use 4 spaces per indentation level instead of tabs[2].

When facing continuation lines[3], you can align elements vertically or using a hanging indent. In the former, you can leverage Python's implicit line joining inside parentheses, brackets, and braces, as follows:

```
# Notice that arguments "num1" and "num3" are vertically aligned
total = sum(num1, num2
            num3, num4)
```

In the case of a hanging indent, there should be no parameters or arguments on the first line after opening the parentheses, brackets, or braces. Then, add a level of indentation. When defining a function, add an extra level of indentation. For example:

[2] You can configure this aspect on the Integrated Development Environment (IDE) you are working in. Jupyter Notebook already does this.

[3] Any statement that requires more than one line can continue in on the next line that is neither a comment or a blank line. The first line of such a statement is known as a *continued line*, while the succeeding ones are *continuation lines* .

```
# Add an indentation level to hanging indents
total = sum(
    num1, num2,
    num3, num4
)

# Add an extra level to function signatures to differentiate
# parameters from the rest of the code
def sum(
        num1: int, num2: int
        num3: int, num4: int) -> int:
    return num1 + num2 + num3 + num4
```

When defining constructs using parentheses, brackets, or braces, you can either line up with the first non-empty character of the previous line:

```
# Line up a list (or any other sequence)
digits = [
    0, 1, 2, 3, 4,
    5, 6, 7, 8, 9
    ]

# Line up the arguments of a function
total = sum(
    num1, num2,
    num3, num4
    )
```

Or you can line up with the character of the continued line (i.e., first line starting the multiline construct):

```
# Line up a list (or any other sequence)
digits = [
        0, 1, 2, 3, 4,
        5, 6, 7, 8, 9
]

# Line up the arguments of a function
total = sum(
        num1, num2,
        num3, num4
)
```

Line Length

Your screen may fit longer lines, but you are not the only one reading the source code. In general, limit the number of characters per line to 80. This limit avoids text wrapping, improving the readability of the code. You can avoid long lines by introducing auxiliary variables, methods, or classes (e.g., to group multiple parameters). If a long line is unavoidable, break it at an appropriate place, and continue on the next line. Use line continuation via its implicit use with parentheses, brackets, and braces, or with an explicit backslash \at the end of the line. The former is preferred over the latter.

```
# Use of line continuation with backslash (\)
with open('path/to/some/file/in/your/computer') as file1, \
     open('path/to/another/file/in/your/computer') as file2:
    content1 = file1.read()
    content2 = file2.read()
```

When breaking code that involves binary operators, prefer to break the line before the operator to increase the understandability of the code:

```
# Break before the binary operator
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Blank Lines

Empty lines provide visual clues about *grouping*. When writing top-level functions and classes, surround them with 2 blank lines. Methods (functions declared within a class) should be surrounded by 1 blank line. Besides these cases, it is good practice to separate functional-related statements with a blank line.

```
# The Python file starts here


# Leave 2 blank lines before and after top-level functions
def square(base: float, exp: float) -> float:
    return base ** exp


# Leave 2 blank lines before and after class definitions
Class Point:
```

```
    # Leave 1 blank line before and after a method definition
    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

Avoid long blocks of statements by introducing auxiliary functions, and avoid having multiple statements on the same line.

```
# Correct
if price > 1000:
    apply_discount()

pay_product(price)
pay_taxes(price)

# Incorrect
if price > 1000: apply_discount()
pay_product(price); pay_taxes(price)
```

String Quotes

There is no preference for the use of single- or double-quotes when defining a string. Both are accepted by Python. It is important, though, to pick one of the two and be consistent in its use. Moreover, when there is a need to write quotes within the string, use the other type of quotes to avoid using backslashes.

```
# Use single-quoted strings
greeting = 'They said: "Hello Karla!"'

# Or double-quoted strings
greeting = "They said: 'Hello Karla!'"
```

Whitespaces

Use whitespaces in the following situations:

- After a comma.
- After a colon, especially when defining a type hint or dictionary.

```
# Correct:
product = {'name': 'tomatoes', 'quantity': 10}
```

```
def sum_integers(numbers: List[int]) -> int:
        return sum(numbers)


# Inorrect:
product = {'name':'tomatoes', 'quantity' :10}

def sum_integers(numbers:List[int]) -> int:
    return sum(numbers)
```

- Before and after `->` to denote the return type of a function, and any binary operator including assignments (`=`), augmented assignments (e.g., `+=`, `-=`), comparisons (`==`, `<`, `>`, `<=`, `>=`, `in`, `is`), and logic connectors (`and`, `or`, and `not`)[4].

```
# Correct:
def sum_integers(numbers: List[int]) -> int:
    return sum(numbers)

total = sum_integers(nums) + 10 - (15 * 40)

# Incorrect:

def sum_integers(numbers: List[int])->int:
    return sum(numbers)

total= sum_integers(nums)+ 10-(15*40)
```

[4] For this course we will **always** keep a space before and after a binary operator, even though the original style guide removes the whitespace when using higher priority operators in an extended expression. This is in line with their suggestion of following "your own judgment" when facing these cases.

- When assigning a value to a keyword parameter (**with** a type hint!).

```
# Correct:
def square(base: float, exp: float = 1) -> float:
    return square_aux(b = base, e = exp)

# Incorrect:
def square(base: float, exp:float=1) -> float:
    return square_aux(b=base, e=exp)
```

- When using colons in an extended slice. However, avoid them when a slice parameter is omitted.

```
# Correct:
ingredients[lower:upper]
```

```
ingredients[lower:upper:]
ingredients[lower::step]
ingredients[:upper:step]
ingredients[lower + offset : upper + offset]

# Incorrect:
ingredients[lower: upper]
ingredients[lower :upper:]
ingredients[lower: :step]
ingredients[ :upper: step]
ingredients[lower + offset:upper + offset]
```

Avoid whitespaces in the following situations:

- After opening or before closing parentheses, brackets, or braces.

```
# Correct:
ingredients = [
    {'name': 'tomatoes', 'quantity': 1},
    {'name': 'onions', 'quantity': 1},
    {'name': 'eggs', 'quantity': 2}
]

# Incorrect:
ingredients = [ { 'name': 'tomatoes', 'quantity': 1 },
                { 'name': 'onions', 'quantity': 1 },
                { 'name': 'eggs', 'quantity': 2 } ]
```

- Between a trailing comma and a close parenthesis.

```
# Correct:
one_item_tuple = (0,)

# Incorrect:
one_item_tuple = (0, )
```

- Before a comma, colon, or semicolon.

```
# Correct:
if age >= 0:
    print('They are', str(age), 'years old')

# Incorrect:
if age >= 0 :
    print('They are' , str(age) , 'years old')
```

- Before an open parenthesis.

```
# Correct:
sum_list([4, 5])

# Incorrect
sum_list ([4, 5])
```

- When trying to align multiple assignment operators.

```
# Correct:
first_name = 'Erik'
last_name = 'Walravens'
age = 23

# Incorrect:
first_name = 'Erik'
last_name  = 'Walravens'
age        = 23
```

- When assigning a value to a keyword parameter (**without** a type hint!).

```
# Correct:
def square(base, exp=1):
    return square_aux(b=base, e=exp)

# Incorrect:
def square(base, exp = 1):
    return square_aux(b = base, e = exp)
```

Trailing Commas

Trailing commas are mandatory when defining a tuple with just one element, in the rest of the cases they are optional. When dealing with tuples, it is a good practice to surround the tuple with parentheses.

```
# Correct:
files = ('config.txt',)

# Incorrect
files = 'config.txt',
```

Redundant trailing commas are helpful when defining a sequence of values that should be extended over time. In these cases, each line has a value with a trailing comma.

```
# Correct:
files = [
    'config.txt',
    'config.properties',
    'requirements.txt',
]

# Incorrect
files = ['config.txt', 'config.properties', 'requirements.txt',]
```

Comments

Comments help you explain the purpose or meaning of your code. Code changes fast, so its comments are prone to be outdated. Ensure you keep them up-to-date every time the code changes. Comments should be complete sentences starting with a # followed by a single space. All sentences within the comment should start with a capital letter. Line length should be respected, thus the rest of the comment should follow its own # in the form of continuation lines. These code comments can also be used as inline comments, which are comments that are written in the same line of a statement. Inline comments should be separated from the statement by at least 2 whitespaces. Obvious comments should always be avoided. Use them only if the code really needs additional explanation.

```
# Correct:

# Only print the message if the age is valid (positive integer).
if age >= 0:
print('They are', str(age), 'years old')

count += 1  # Skip first item of the sequence


# Incorrect:

#check if the age is greater or equal to 0
if age >= 0:
print('They are', str(age), 'years old')

count += 1#add 1
```

Docstrings

You should write documentation for all public modules, functions, classes, and methods. This documentation is known as **docstring** in Python, and they have been defined in the PEP 257 document. Besides these conventions, you can also follow other formats to write your docstrings. In this course, we use the **reST** (reStructuredText) format, which is used by default in the JetBrains PyCharm IDE. Have a look at both documents and follow their guidelines when writing your docstrings. You can also opt for any other format as long as you stay consistent in its use along your code.

In general, docstrings start with a triplet of double quotes, followed by the description of the construct, parameters or field descriptions, and description of the return value (in the case of non-void functions and methods). Prefer to start with a verb that describes the action performed by the construct in case of dealing with functions or methods.

```
# Correct:
def sum(num1: int, num2: int
        num3: int, num4: int) -> int:
    """
    Returns the addition of four integer numbers received as
    parameters.

    :param num1: first integer number
    :param num2: second integer number
    :param num3: third integer number
    :param num4: fourth integer number
    :returns: the addition of the four numbers.
    """
    return num1 + num2 + num3 + num4


# Incorrect:
def sum(num1: int, num2: int
        num3: int, num4: int) -> int:
    #does something with the numbers and maybe it returns
    #something else.
    return num1 + num2 + num3 + num4
```

References

© 2021-2023. **VU Amsterdam - TU/e**