Universidad Carlos III
Software Development 2023-24
Course 2023-24

Coding Standards

Date: **14/03/2024**
GROUP: **88**

Members:

100495878 - Diego San Román Posada

100495857 - Bárbara Sánchez Moratalla

100495774 - Izan Sánchez Álvaro

Index

NAMES

1. File names

Definition:
The rule specifies the naming convention for files, requiring them to follow PascalCase or camelCase and include appropriate file extensions based on their content.

Guideline:

- File names must adhere to PascalCase or camelCase format, where each word in the name begins with a capital letter and no spaces are allowed.
- File names should end with the appropriate file extension indicating the type of content. For Python files, use ".py"; for text files, use ".txt", and so on

Example of correct usage:

- HotelManager.py
- test.json
- main.py
- HotelManagementException.py

Example of incorrect usage:

- Hotel_Manager.py
- test
- MAIN.py
- Hotel Management Exception.py

2. Variable names

Definition:
The objective of this rule is to establish clear regulations and restrictions in relation with the naming of variables. Its primary aim is to ensure a clear view and understanding across the workspace, making users able to distinguish between variables and constants. Additionally, it also serves as a common guideline for the declaration of variables and constants.

Guideline:

- Any time you want to declare a variable, it should always start with lowercase, except constant variables.
- Any constant variable's name should be written in uppercase.
- If the name of a variable consists of more than one word, the method to use is "camelCase".

Example of correct usage:

- Constant:

```
DATA = 'hello'
```

- Variable: (1) normal case, (2) snake_case, (3) camelCase

```
variable1 = 2  //(1)
example_variable = 4 //(2)
exampleVariable = 6 //(3)
```

Example of incorrect usage: (1) constant, (2) variable

```
data = 'hello' //(1)

Variable1 = 2  //(2)
```

- (1) is incorrect because since it is a constant variable it should be written in UpperCase.
- (2) is incorrect because a variable cannot start with UpperCase.

3. Functions vs method names

Definition:
This rule is very similar to the one described above (6). Its primary objective is to facilitate the establishment of a consistent structure within codebases, particularly in regard to the definition of functions and methods within classes. In addition, it tries to ensure clarity and differentiation between various elements, such as functions and methods, so that there is a coherent organizational framework for all users and developers operating within the codebase.

Guideline:

- When defining a function we should name it in lowercase and, exceptionally, if the name consists of more than one word, with the "camelCase" format.
- The name of either a function or a method must never start with UpperCase.
- When defining a method we should use the same criteria as if it were a function, but in case the name consists of more than one word, we should use "snake_case" instead of "camelCase".

Example of correct usage:

```
# Function

def sum(x, y):
    return x+y

def sum_floats(x, y):
    return float(x) + float(y)

# Method

class Example:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def sumFloats(self):
        return float(self.x) + float(self.y)
```

Example of incorrect usage:

```
# Function

def Sum(x, y):
    return x+y

def sumFloats(x, y):
    return float(x) + float(y)

# Method

class Example:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def sum_floats(self):
        return float(self.x) + float(self.y)
```

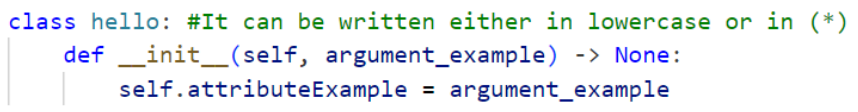
4. Class/Module names

Definition:
To maintain a uniform coding standard so that all users have a guideline on how to program in the same codespace, we continue on stating how the different items of the coding should be named when declaring and initializing them. In this case, we propose a naming convention for the creation of classes or modules.

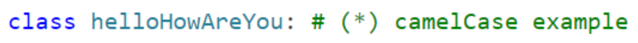
Guideline:

- When defining a module/class we should name it in lowercase or using "camelCase" if the name contains more than one word.
- The name of the arguments passed to the class should be written in lowercase or in "snake_case".
- In the same way, the name of the attributes inside the class (self.name) should be written in lowercase or in "camelCase".

Example of correct usage:

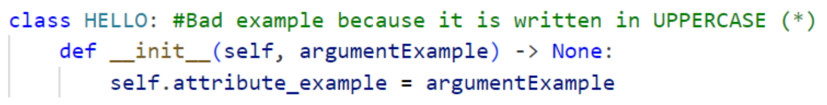


```
class hello: #It can be written either in lowercase or in (*)
    def __init__(self, argument_example) -> None:
        self.attributeExample = argument_example
```

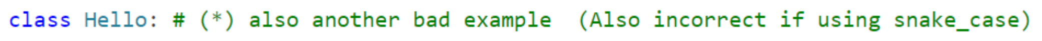


```
class helloHowAreYou: # (*) camelCase example
```

Example of incorrect usage:



```
class HELLO: #Bad example because it is written in UPPERCASE (*)
    def __init__(self, argumentExample) -> None:
        self.attribute_example = argumentExample
```



```
class Hello: # (*) also another bad example  (Also incorrect if using snake_case)
```

5. Inappropriate Variable Names

Definition:
Variable names should be chosen thoughtfully to avoid any offensive, derogatory, or inappropriate language. It's essential to maintain professionalism and respect for all individuals who may read or interact with the codebase. Inappropriate variable names can lead to discomfort, offense, or even legal consequences.

Guideline:
Select variable names that are neutral, respectful, and free from any language that could be considered offensive, discriminatory, or insensitive. Avoid using terms that could be interpreted as derogatory, discriminatory, or disrespectful towards individuals or groups based on factors such as race, gender, ethnicity, religion, sexual orientation, disability, or any other protected characteristic.
When in doubt about the appropriateness of a variable name, it is better to change it and choose a more neutral and respectful alternative.

Example of correct usage:

```
a = 5
b = 3
name = “Santiago”
```

Example of incorrect usage:

```
gilipollas = 5
imbécil = 3
tonto = “Santiago”
```

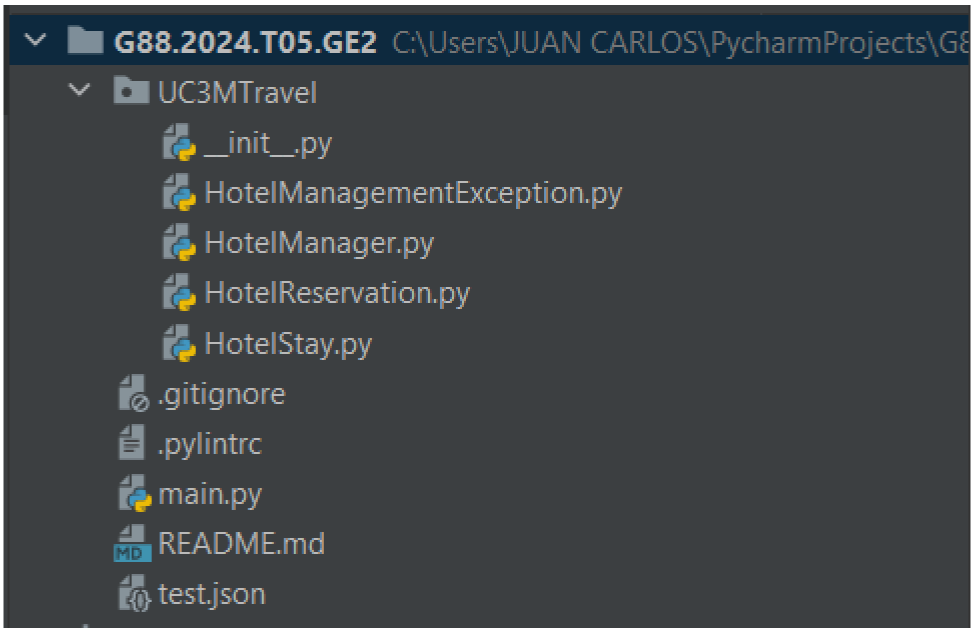
STRUCTURE

6. Directory organization

Definition:
The rule establishes the way in which python files should be organized in a directory. Only the main file should appear in the directory, while all the others should be kept in an auxiliary directory.

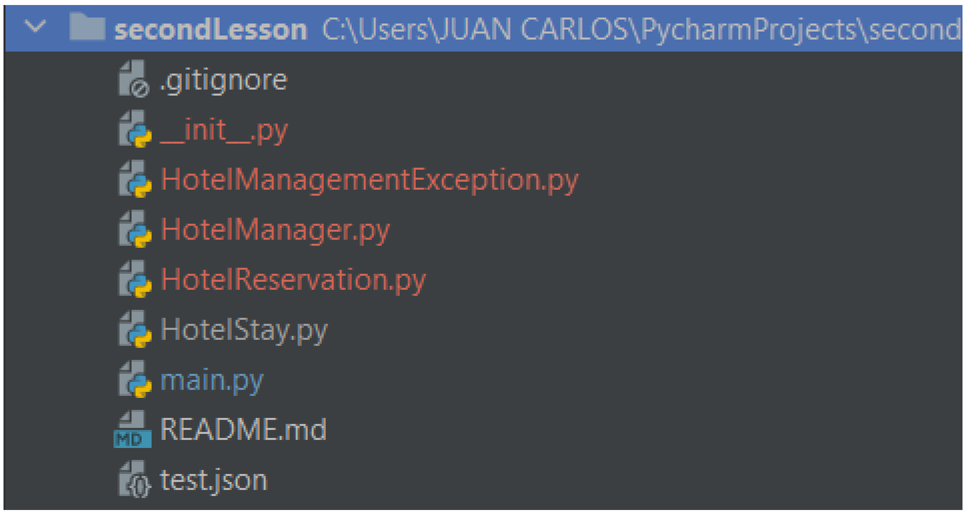
Guideline:
- Only the main python file (under the name main.py) must appear in the "general" directory.
- Auxiliary python files containing classes must be saved in a sub-directory and imported by the main.py file at the beginning of the code.

Example of correct usage:



```
G88.2024.T05.GE2 C:\Users\JUAN CARLOS\PycharmProjects\G8
    UC3MTravel
        __init__.py
        HotelManagementException.py
        HotelManager.py
        HotelReservation.py
        HotelStay.py
    .gitignore
    .pylintrc
    main.py
    README.md
    test.json
```

Example of incorrect usage:



```
secondLesson  C:\Users\JUAN CARLOS\PycharmProjects\second
    .gitignore
    __init__.py
    HotelManagementException.py
    HotelManager.py
    HotelReservation.py
    HotelStay.py
    main.py
    README.md
    test.json
```

COMMENTS

7. Comments structure

Definition:
This rule defines the structure of comments in Python. It specifies that multi-line comments should be enclosed within triple double quotes (""") instead of using the pound symbol (#).

Guideline:

- When a comment spans more than one line, it should be enclosed within triple double quotes (""") for clarity and consistency. Single-line comments can continue to use the pound symbol (#).
- Long comments exceeding the maximum line size stated should be enclosed within triple double quotes for readability and consistency.
- The structure of the short comments must be the following:
  # short comment
- The structure of the long comments must be the following:
  “””
  long comment
  “””

Example of correct usage:

```
"""
This is a long comment. It is enclosed within triple double
quotes and indented properly
for a better readability and consistency.
"""
# This is a one line comment.
```

Example of incorrect usage:

```
# This comment exceeds the maximum line size stated and should be
# enclosed within triple double quotes for readability and
# consistency.
“””This comment is short”””
```

ADDITIONAL IMPLEMENTATIONS

8. Copyright start

Definition:
The rule defines the required format for copyright notices at the beginning of Python code files. It mandates the inclusion of specific information such as the author's name, the date the code was written, and the institution, in this case, the University Carlos III of Madrid.

Guideline:
The copyright note must be placed at the beginning of each file.
The note must include the following information:

- Copyright symbol (©)
- Year of creation or modification
- Author's name
- Institution associated with the author (University Carlos III of Madrid)

Example of correct usage:

```
"""
Copyright © 2024 Author Name
Written on Actual Date
University Carlos III of Madrid
"""
```

Example of incorrect usage:

```
"""
Copyright © 2024
Written on Actual Date
Leganés, Madrid, Spain
"""
```

9. Maximum Line Size

Definition:
The maximum line size rule defines the maximum allowed length for a single line of code within the source file. It aims to promote code readability and maintainability by preventing excessively long lines that may be difficult to understand or navigate.

Guideline:
- Avoid excessively long lines of code as they can make the code harder to read and understand, especially when viewed on devices with limited screen width or in code review tools.
- Consider breaking long lines into multiple shorter lines using appropriate line continuation techniques, such as parentheses for expressions, backslashes for multiline strings, or breaking function arguments onto multiple lines.
- Ensure that breaking long lines does not sacrifice code clarity or introduce ambiguity. Aim to maintain logical groupings and readability when splitting lines.

Example of correct usage:
```
long_string = "This is a very long string that " \
              "needs to be split into multiple lines " \
              "for readability."
```

Example of incorrect usage:
```
long_string = "This is a very long string that needs to be split into multiple lines for readability but it is not split into those multiple lines."
```

10. Maximum number of arguments for a function/method

Definition:

This rule is related to the maximum number of arguments that a function can have. To ensure the functions make sense, we define 8 as the maximum number of arguments that a function can have. This rule serves as a crucial guidance for maintaining uniformity inside the program.

Guideline:

- Each time we define a new function, we have to make sure that it does not have more than 10 arguments. In the case of methods inside of a class, the self argument counts as one arguments.

Example of correct usage:

```
def roomReservation(self, credit_card, name_surname, id_card,
                    phone_number, room_type, arrival_date,
                    num_days):
```

Example of incorrect usage:

```
def roomReservation(self, credit_card, name_surname, id_card,
                    phone_number, room_type, arrival_date,
                    num_days, more_arguments, more, more):
```

11. Maximum number of attributes for a class

Definition:
This rule is related to the maximum number of attributes that a class can have. To ensure the functions make sense, we define 8 as the maximum number of attributes that a class can have. This rule serves as a crucial guidance for maintaining uniformity inside the program.

Guideline:

- Each time we define a new class, we have to make sure that it does not have more than 10 attributes.

Example of correct usage:
```
class hotelReservation:
   def __init__(self, id_card, credit_card, name_surname,
               phonenumber, room_type, arrival_date,
               numdays):
```

Example of incorrect usage:
```
class hotelReservation:
   def __init__(self, id_card, credit_card, name_surname,
               phonenumber, room_type, arrival_date,
               numdays, more, more, more, more, more, more):
```

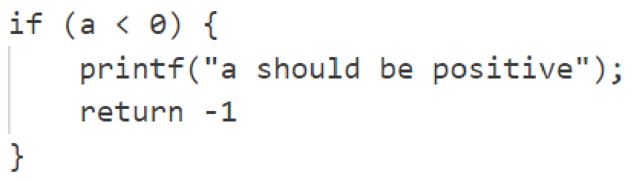
12. Indentation

Definition:
This rule is related to the internal structure of the program. To ensure a consistent organization, a standard indentation of 4 spaces is imposed in the code. This rule serves as a crucial guidance for maintaining uniformity inside the program.

Guideline:

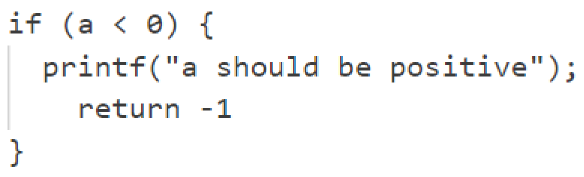
- Each time we start writing code after the creation of a class, function, method or condition; we have to care about maintaining the indentation of 4 spaces inside.

Example of correct usage:



```
if (a < 0) {
    printf("a should be positive");
    return -1
}
```

Example of incorrect usage:



```
if (a < 0) {
  printf("a should be positive");
    return -1
}
```

13. If __name__==__main__

Definition: This condition is used to verify that the user is running a script and not an imported file. By including it in our main script (main.py) and not in the other files, we indicate clearly which is the file meant to be run.

Guideline:

- A condition If __name__==__main__ must be included in the main.py code to verify that the code we are running is not imported from somewhere else.
- Therefore, by including this line of code, we indicate to the reader that this is the main script and that it is the file meant to be run.
- The main content of our code must be written inside this if block or, preferably, invoked by calling the main() function.
- All the other files should not contain this if statement. Hence implying that they are solely meant to be imported.

Example of correct usage:

```python
from Geometry import Square

def main():
    square1 = Square(10)
    print("Area: ", square1.area)

if __name__ == "__main__":
    main()
```

Example of incorrect usage:

```python
# Example 1, being at the main.py file...
from Geometry import Square

square1 = Square(10)
print("Area: ", square1.area)   # At first sight it is not that clear if the user is supposed to run this file.


# Example 2, being at the Square.py file...
def main():
    class Square:
        def __init__(self, side):
            self.__side = side
        @property
        def side(self):
            return self.__side

        @side.setter
        def side(self, value):
            if value <= 0:
                raise ValueError("Side size must be positive.")
            self.__side = value

        def area(self):
            return self.__side**2


if __name__ == "__main__":
    main()
```

14. String Quotes

Definition: This rule defines the preference of double quotes (“x”) over single quotes (‘x’) to delimit strings.

Guideline:

- When writing strings, the text must be delimited by double quotes at the beginning and at the end of it.

Example of correct usage:

```
print("This string quoting is correct.")
print("This option's correct too.")
print("Another valid possibility: 'using single string-quotes within the string'.")
```

Example of incorrect usage:

```
print('This string quoting is not correct (although valid).')
print("This string is delimited by different quotation marks and therefore it is not valid.')
print('This option's not correct because it uses the quotation marks to delimit the string inside the string itself.')
print("Similarly: "this is not correct either".")
```

15. Data Encapsulation

Definition: This rule is directed towards Object-Oriented Programming and defines the need of encapsulating data into classes. This way, we ensure that data is only accessed and modified when using predefined methods and, thus, we improve security and robustness.

Guideline:

- Every object must be defined by its corresponding class.
- Inside each class, there must be:
    - An __init__() method initializing the object attributes.
    - Getter and setter methods defined to access private attributes and to make sure that any value introduced is valid.

Example of correct usage:

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius  # We ensure that the attribute is
private

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive.")
        self._radius = value

circle = Circle(5)
print("Radius:", circle.radius)
```

Example of incorrect usage:

```
class Circle:
      def __init__(self, radius):
            self.radius = radius  # Public attribute
```

```
# No getter and setter is defined and, thus, there is no validation of data. A user can introduce a negative radius and no error will pop up.
    def diameter(self):
        return 2 * self.radius

circle = Circle(-5)
print("Radius:", circle.radius)
print("Diameter:", circle.diameter())
```

16. Magic Methods

Definition: This rule, meant for Object-Oriented Programming too, defines the need of implementing magic methods for the initialization and representation of objects.

Guideline:

- All classes must implement the ‘__innit__’ initialization method, in which they initialize all attributes related to the object.
- Other magic methods, such as ‘__str__’ for defining what to print are also recommended.

Example of correct usage:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "Point: " + str(self.x) + ", " + str(self.y)

myPoint = Point(3, 2)
print(myPoint)  # Output is Point: 3, 2
```

Example of incorrect usage:

```
# Example 1
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

myPoint = Point(3, 2)
print(myPoint)  # Output is <__main__.Point object at 
0x0000021E04633B80>

# Example 2
class Point:
    def __init__(self, x, y):
        self.x = x

    def __str__(self):
        return "Point: " + str(self.x) + ", " + str(self.y)
```

```
myPoint = Point(3, 2)
print(myPoint)  # Attribute y was not initialized, so we get an
error
```