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.

Consistency

1. Be consistent when using the freedom that this standard leaves.

Indentation

2. *Always* indent *systematically* a multiple of 4 spaces. *Never* use TAB characters in source code. (Let your editor expand the TAB key to spaces. Jupyter Notebook already does this)

Line length

3. *Always* limit the line length to at most 80 characters. (Set a right margin.)

Empty lines

4. *Always* use one empty line before and after the following cases:

    - functions (Week 3)
    - classes (Week 5) are surrounded by 2 white lines instead

    White lines can be used to separate *groups* of statements or assignments to enhance readability

Spacing 1

5. *Never* write a space before and *always* write one space after the following items (*unless* at line end):

    - `,` `:`

Spacing 2

6. *Always* write one space before and after the following items (*unless* at line begin/end):

    - keywords: `if` `for` `while` etc.
    - binary operators (*except* `.`): `=` `+` `-` `*` `/` `%` `==` `!=` `<` `>` `<=` `>=` `&&` `||` etc.

Comments

7. *Always* explain each variable declaration in a comment.

Docstring

8. *Always* specify each public entity (Classes and functions) in a docstring comment.

Naming 1

9. Variable, function and class names should *always* reflect the usage rather than the implementation.

Naming 2

10. *Always* use the associated name conventions for different objects:

    - Use CapWords for class names
    - Use lowercase with words separated by underscores for function and variable names
    - Use All caps with words separated by underscores for the name of constants

Type Hints

11. *Always* add type hints to functions using ':' or '-¿'

BAD

```
def RUN(): #This is the part that does things
    x1= input() #These lines save the input
    y1= input()
    x2= input()
    y2= input()
    x3= input()
    y3= input()
    if((x1>x2) or (y1<y2)): #This part checks if the rectangle is well-de
      print("error")
    elif(((x3>=x1) and (x3<=x2)) and ((y3<=y1) and (y3>=y2))): #This chec
       print("inside")

    else :
      print("outside") #If the point is not in the rectangle it is outsid
```

GOOD

```
def is_in_rectangle(left_coord: float, right_coord: float,
                    bottom_coord: float, top_coord: float,
                    point_x: float, point_y: float) -> None:
    """ Checks if a given point is inside a given rectangle

    Prints either 'error', 'inside' or 'outside'
    """
    # If the rectangle is poorly defined, print 'error'
    if ((left_coord > right_coord) or (bottom_coord < top_coord)):
      print("error")
    # If the point is in the rectangle, print 'inside'
    elif (point_x >= left_coord) and (point_x <= right_coord) and
        (point_y <= top_coord) and (point_y >= bottom_coord):
      print("inside")
    # Else the point is out of the rectangle, print 'outside'
    else:
      print("outside")

# Get the rectangle and point input
rectangle_left = input('Left side coordinate of the rectangle = ')
rectangle_right = input('Right side coordinate of the rectangle = ')
rectangle_bottom = input('Bottom coordinate of the rectangle = ')
rectangle_top = input('Top coordinate of the rectangle = ')
point_X = input('X coordinate of the point = ')
point_Y = input('Y coordinate of the point = ')

# Run the function with the given parameters
check_rectangle(rectangle_left, rectangle_right, rectangle_bottom,
                rectangle_top, point_X, point_Y)
```

Notes

Well-organized source code is important for several reasons.

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

Here is some further background information on each of the rules.

Consistency

1. Consistency especially plays a role in the placement of opening braces . Either place them at the end of the line with the controlling statement, or at the beginning of a line by themselves directly below the the controlling statement. An advantage of the latter style is that opening and closing braces are vertically aligned. A disadvantage is that it takes more vertical space.

Indentation

2. Indentation provides visual clues about the containment structure (*nesting*). One or two space indentation does not provide enough visually guidance. Some standards prescribe indenting by multiples of three spaces (because that interferes with TAB characters, thereby discouraging them even more). Indenting by more than four spaces is a waste, and leaves less room in view of the line length limit (also see the next note).

Line length

3. Your screen may fit longer lines, but you are not the only one reading the source code. Moreover, long lines are hard to parse. Also see next note.

   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.

   Of course, the line length of generated code may not be under your control.

Empty lines

4. Empty lines provide visual clues about *grouping*, on an intermediate level (also see next two notes). Besides the situations mentioned in Rule 4, it is good to delineate *groups of related statements* by empty lines; for example, needed loop variable declarations, the loop, and finalization of the loop. By the way, such grouping can also be made explicit by *brackets*, defining a *block*, possibly with its own local variables.

   Avoid long blocks of statements by introducing auxiliary methods (Extra functions).

Spacing 1

5. Spacing also provides visual clues about *grouping*, but on a lower level than

empty lines (see preceding note; also see next note). This rule concerns *punctuation*.

Commas are used to separate items in lists, such as *parameters* (both formal and actual), and expressions.

There should never be multiple *colons* on the same line.

Spacing 2

6. Spacing improves readability, especially when quickly scanning source code, rather than reading it slowly in full detail.

   Readability of expressions can be further improved by appropriate use of *parentheses*, and *auxiliary variables and functions*.

Comments

7. Variables are introduced for a specific purpose. The name of the variable should reflect that purpose. However, a name should also not be too long. Furthermore, the purpose usually involves relationships to other elements of the program. A comment makes this explicit. To avoid having to spend any effort on deciding whether to include a comment or not, the rule is simply to provide it always.

   It is also good practice to provide comments with non-obvious statements. However, there is such a thing as *superfluous* comments. Do not comment the obvious.

Docstring

8. Public entities are typically classes, methods, functions and (non-local) constants. Public entities can be used anywhere in a program. Therefore, their usage should be well documented. Docstring comments have two benefits over ordinary (non-docstring) comments:

   - They support additional features, such as *tags*, to structure documentation.
   - They can be extracted from the source code and presented separately, as a document with cross references.

   Many Python Integrated Development Environments (IDEs) provide additional benefits when using docstring comments. Docstring comments can be identified by the triple quotes and often span multiple lines

   ””” This is a docstring ”””

Naming

9. Variables, Functions and Classes are introduced for a specific purpose. The name of the variable should reflect that purpose. But names can also give visual clues to the usage. Using different types of names helps with quickly interpreting code.

   - 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 convention (E.g. KittenShop).

- Functions and variables should be written in lowercase with words separated by underscores. For functions it is also allowed to use camelCase, where the first word starts with a lowercase letter but the other words start with uppercase (E.g. kittenShop), but all lowercase is preferred.
- Constants are written in all caps to remind developers that these values should not and will not be changed (E.g. MAXVALUE).

Type Hints

10. Functions often only allow a specific type of variables to be given as parameters. Because code is often created in (large) groups, others may not know what variable types are allowed in a function and which ones are not. Type hints help developers so that they don't have to carefully look through (complex) functions to be able to use these functions.

    By adding type hints ...

    - to the function, a coder can quickly know what type the function returns and use the function without exactly knowing what the outcome could be.
      Example:

      ```
      def example(var1) -> TYPE:
      ```

    - to the parameters, someone can quickly see what types the variables should be, before they can successfully use the function.
      The only parameter that doesn't need a type hint is the parameter 'self'. This parameter is standard for methods but does not need an explicit type hint. (This is used in classes).
      Example:

      ```
      def example(var1: TYPE):
      ```

    Note: It is rumoured that Python might update at some point to change type hints from voluntary to mandatory.

References

[1] *Style Guide for Python Code*. PEP 8, 2001.
`https://www.python.org/dev/peps/pep-0008/`