Python Coding Standards Documentation

Version:	1.1.0
Status:	Released
Date:	21/4/2024

Prepared by:			
Name	**Metric number**	**Department**	**Date**
Lutfil Hadi Bin Abdul Rahim	U2000499	QA (Standard & Compliance)	20/4/2024
Bryan Phang Yik Loong	U2102738	QA (Standard & Compliance)	20/4/2024
Approved by:			
Name	**Metric number**	**Department**	**Date**
Released by:			
Name	**Metric number**	**Department**	**Date**

Version History

Version	Release Date	Section	Amendments
1.0.0	15/4/2024	All	Original Document
1.1.0	21/4/2024	All	Updated original document content

Change Record

Version	Date	Affected Section	Change Request #	Reason/Initiation/Remarks
1.0.0	15/4/2024	-	-	Request review by other departments
1.1.0	20/4/2024	2.0 Related documents	1	Update the references and glossary
		3.3 Documentation and Docstrings	1	Update the standards of the docstrings
		3.4 Imports and Packages	1	Include the standards for packages

Table of Contents

1.0 Description

1.1 Purpose

This document, primarily through reference to establish Python ‘community” documents, specifies standards and best practices for coding and documenting Python code for the development of the Optical Character Recognition (OCR) System for Handwritten Multiple-Choice Question (MCQ) Answer Sheet Evaluation project.

This document outlines the coding standards, design principles, and naming conventions specifically for Python development projects. It serves as a reference for the software development teams involved in this project, facilitating the creation of efficient, maintainable and consistent code.

1.2 Scope

All Python code checked into the OCR software-development project’ permanent repository shall adhere to the standards and best practices specified in this document and its references.

2.0 Related Documents

2.1 References

Rossum, G. V., Warsaw, B., Coghlan, A. (2013). *PEP 8 – Style guide for Python Code.* https://peps.python.org/pep-0008/

8. Errors and Exceptions. (n.d.). Python Documentation. https://docs.python.org/3/tutorial/errors.html

Goodger, D., Rossum, G. V. (2001). *PEP 257 – Docstring Conventions.* https://peps.python.org/pep-0257/

Goodger, D. (2001) *PEP 256 – Docstring Processing System Framework.* https://peps.python.org/pep-0256/

2.2 Abbreviations and Acronyms

Abbreviation/Acronym	Description
PEP	Python Enhancement Proposal, a standard for documentation within the Python community

2.3 Glossary

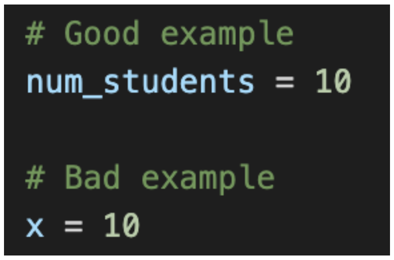
Glossary	Description
docstring	A string literal that occurs as the first statement in a module, function, class, or method definition

3.0 Code Standards

3.1 Naming Convention

3.1.1 Descriptive Names

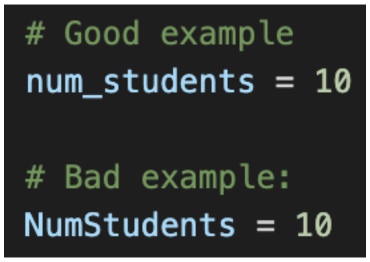
The named assigned shall be meaningful and accurately describes the purpose or role of a variable, function, class, or module.



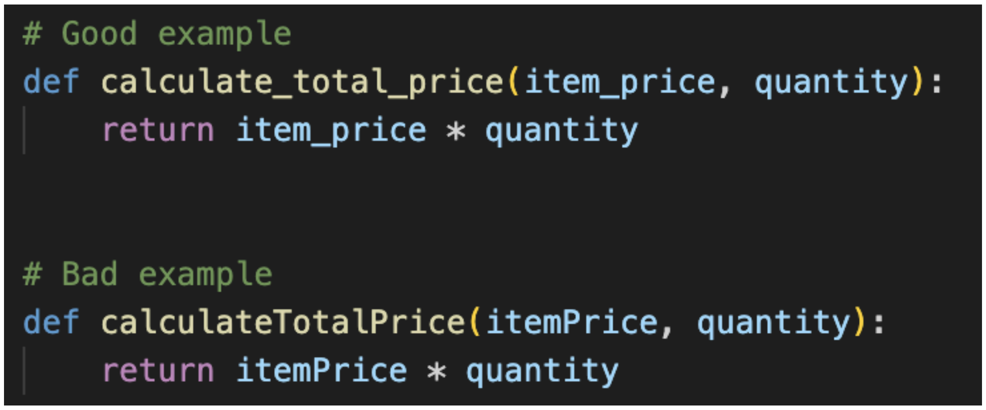
```python
# Good example
num_students = 10

# Bad example
x = 10
```

3.1.2 “snake_case” for variable and function names

Variables and functions shall follow the snake_case convention, which the words in lowercase form and/or separated by underscores (_).



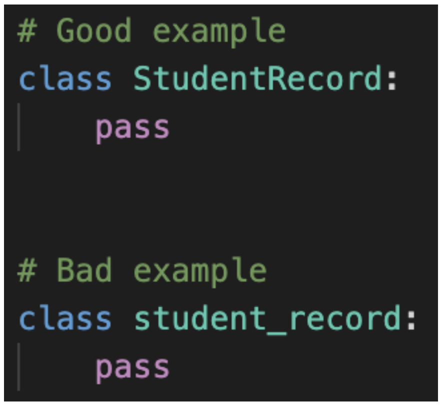
```python
# Good example
num_students = 10

# Bad example:
NumStudents = 10
```



```python
# Good example
def calculate_total_price(item_price, quantity):
    return item_price * quantity


# Bad example
def calculateTotalPrice(itemPrice, quantity):
    return itemPrice * quantity
```

3.1.3 “CamelCase” for class names

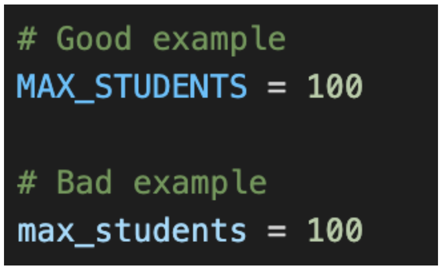
Class names shall follow the “CamelCase” conventions, which the words start with an uppercase letter and/or with no underscores between the words.



```python
# Good example
class StudentRecord:
    pass


# Bad example
class student_record:
    pass
```

3.1.4 Uppercase for constants

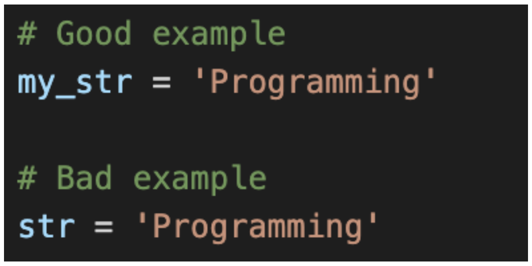
All constants shall be differentiated from other variables, with the all the words in uppercase and/or separated by underscores.



```python
# Good example
MAX_STUDENTS = 100

# Bad example
max_students = 100
```

3.1.5 Reserved keywords

Python reserved keywords shall be avoided to be used as identifiers.

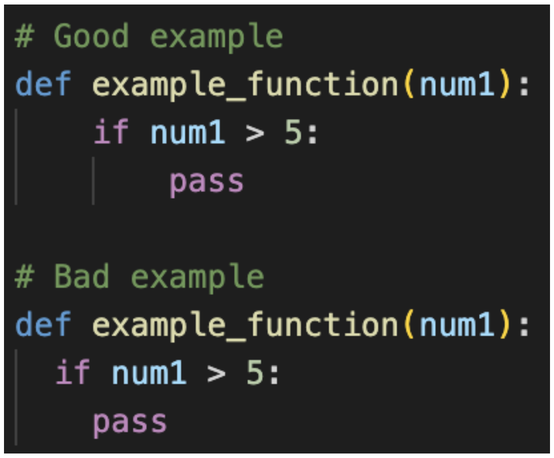


```python
# Good example
my_str = 'Programming'

# Bad example
str = 'Programming'
```

3.2 Code Formatting

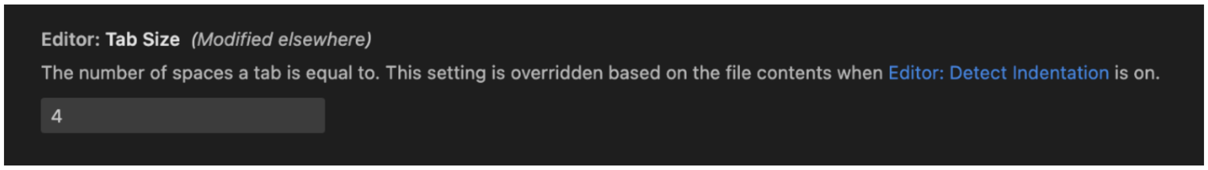
3.2.1 Indentation

Four spaces shall be used for each level of indentation



```python
# Good example
def example_function(num1):
    if num1 > 5:
        pass

# Bad example
def example_function(num1):
  if num1 > 5:
    pass
```

To change tab size in Visual Studio Code, proceed to **Settings** > **Commonly Used** > **Editor: Tab Size**

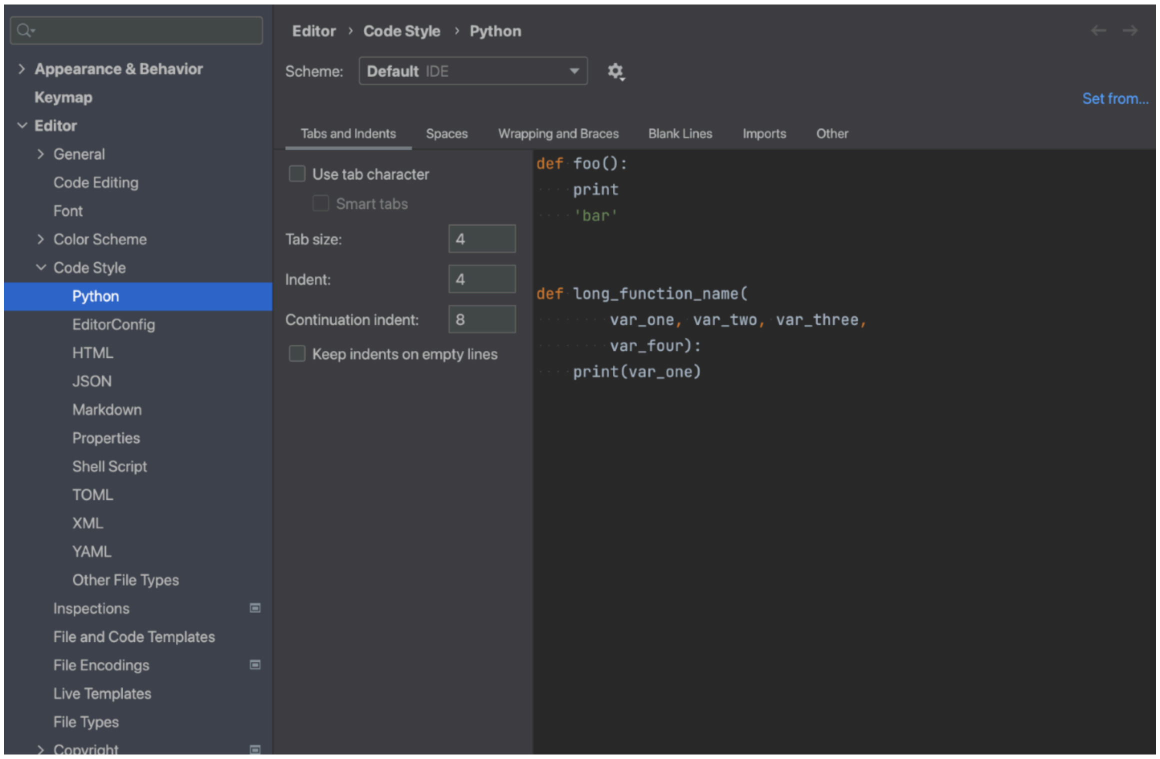


Editor: **Tab Size** *(Modified elsewhere)*

The number of spaces a tab is equal to. This setting is overridden based on the file contents when Editor: Detect Indentation is on.

4

To change tab size in PyCharm, proceed to **Settings** > **Editor** > **Code Style** > **Python** > **Tab size**

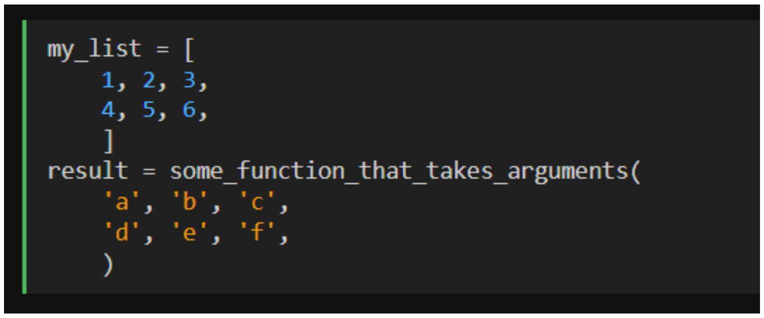


Scheme: Default IDE

Use tab character: unchecked
Smart tabs: unchecked
Tab size: 4
Indent: 4
Continuation indent: 8
Keep indents on empty lines: unchecked

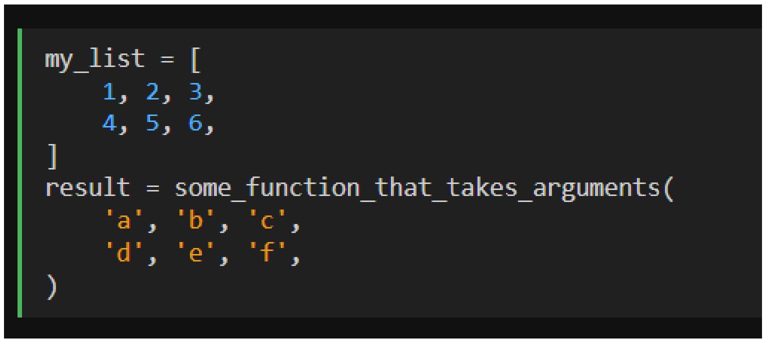
```
def foo():
    print
    'bar'


def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

For braces/brackets/parenthesis alignment, the closing brace/bracket/parenthesis shall be either aligned under the first non-whitespace character of the last line of the construct, as in:



```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

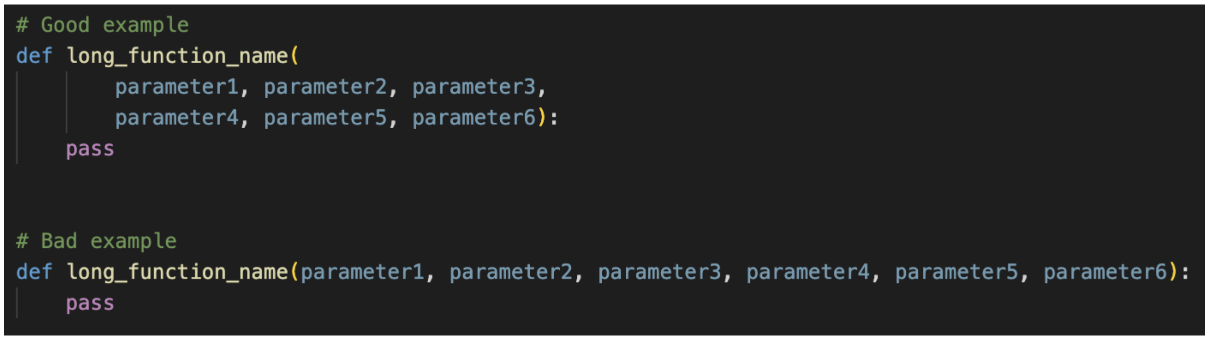
The braces/brackets/parenthesis may also line up under the first character of the line that starts the construct.



```python
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

3.2.2 Line Length

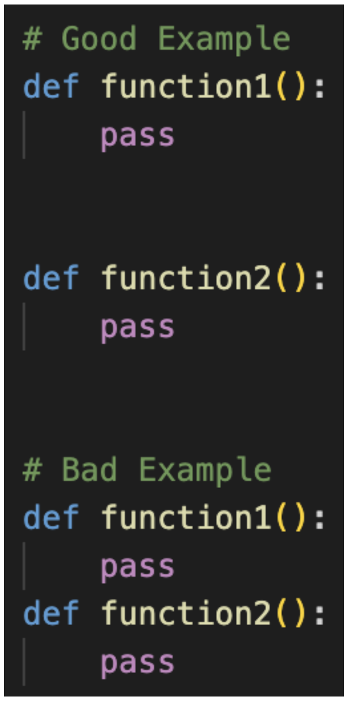
The lines of code shall be kept to a maximum of 79 characters per line of code. If the line of code is longer or exceeded the set number of characters, the code shall be broken into multiple lines, adhering to logical indentations.



```python
# Good example
def long_function_name(
        parameter1, parameter2, parameter3,
        parameter4, parameter5, parameter6):
    pass


# Bad example
def long_function_name(parameter1, parameter2, parameter3, parameter4, parameter5, parameter6):
    pass
```

3.2.3 Blank Lines

Blank lines shall be used to separate logical sections of a code, such as functions, classes, or blocks of related code. Two blank lines shall be used between top-level definitions instead.



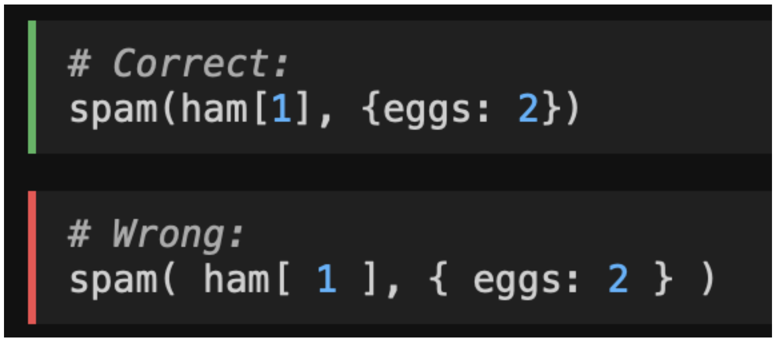
```python
# Good Example
def function1():
    pass


def function2():
    pass


# Bad Example
def function1():
    pass
def function2():
    pass
```

3.2.4 Whitespaces

3.2.4.1 Avoid Extraneous Whitespace

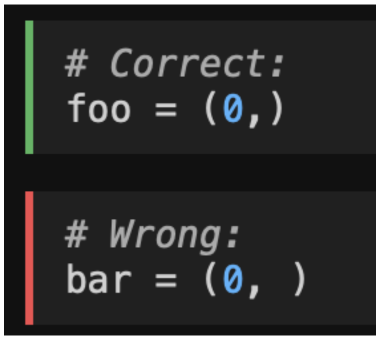
Avoid unnecessary spaces in the following scenarios:

- Within parentheses, brackets, or braces



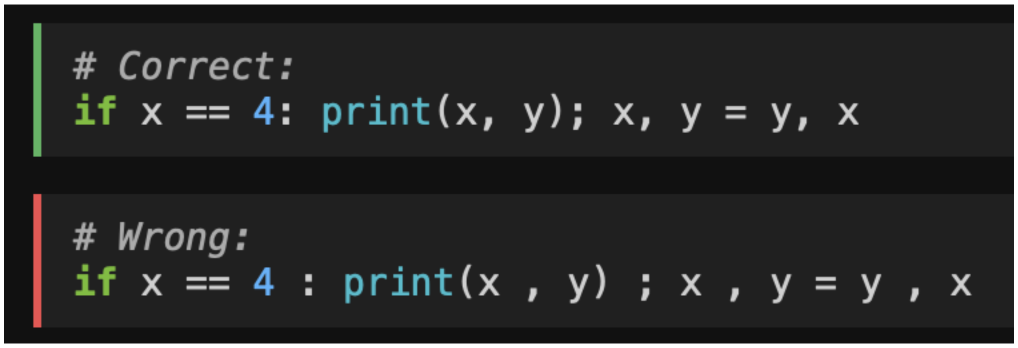
```python
# Correct:
spam(ham[1], {eggs: 2})

# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parentheses



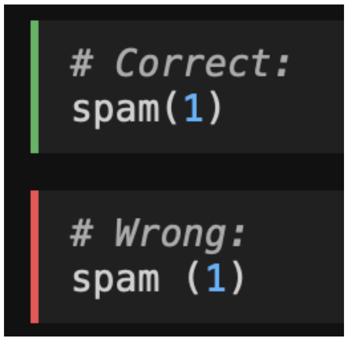
```python
# Correct:
foo = (0,)

# Wrong:
bar = (0, )
```

- Before a comma, semicolon, or colon



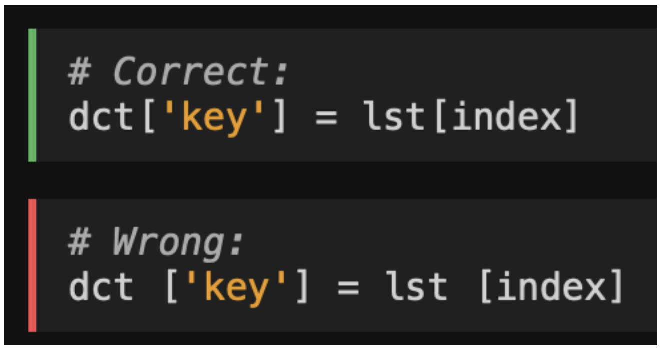
```python
# Correct:
if x == 4: print(x, y); x, y = y, x

# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

- Before the open parentheses that starts the argument list of a call function



```python
# Correct:
spam(1)

# Wrong:
spam (1)
```

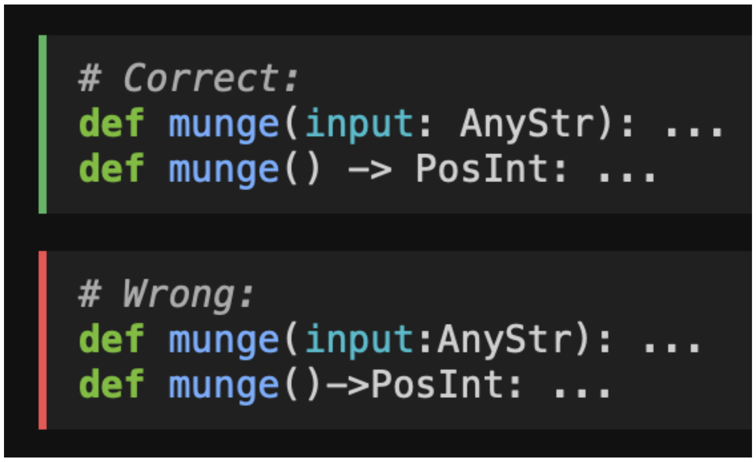
- Before the open parentheses that starts an indexing or slicing



```python
# Correct:
dct['key'] = lst[index]

# Wrong:
dct ['key'] = lst [index]
```

3.2.4.2 Other Recommendations

- Remove trailing whitespace as it can be misleading and is not preserved in all environments.
- Maintain single spaces around assignments, comparison, and Boolean operators.
- Follow standard colon rules for function annotations and ensure spaces around the arrow (->) if present



```python
# Correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...

# Wrong:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

- Use spaces around the equal sign (=) when combining argument annotations with default values

3.3 Documentation and Comments

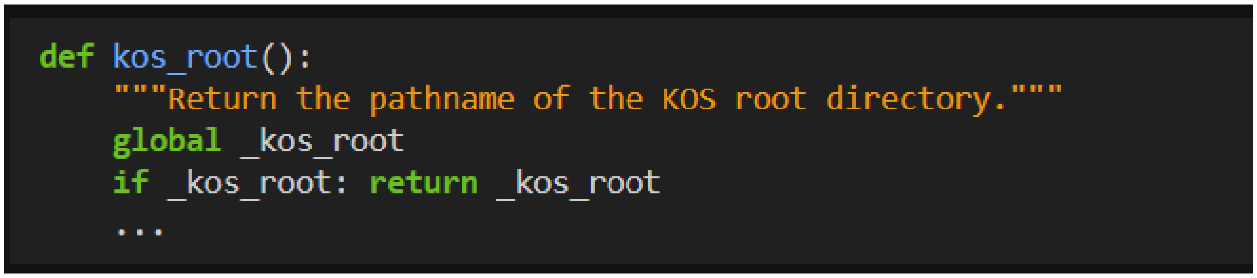
3.3.1 Documentation Strings (docstrings)

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.

Docstrings shall be written for all public modules, functions, classes, and methods to describe what the methos does. The comments should appear after the “def” line.

One-line Docstring

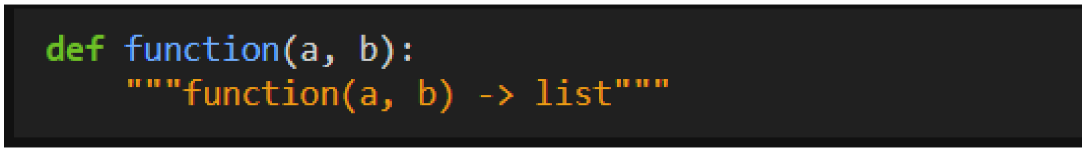
Example of one-line docstrings:



```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Notes for one-line docstrings:

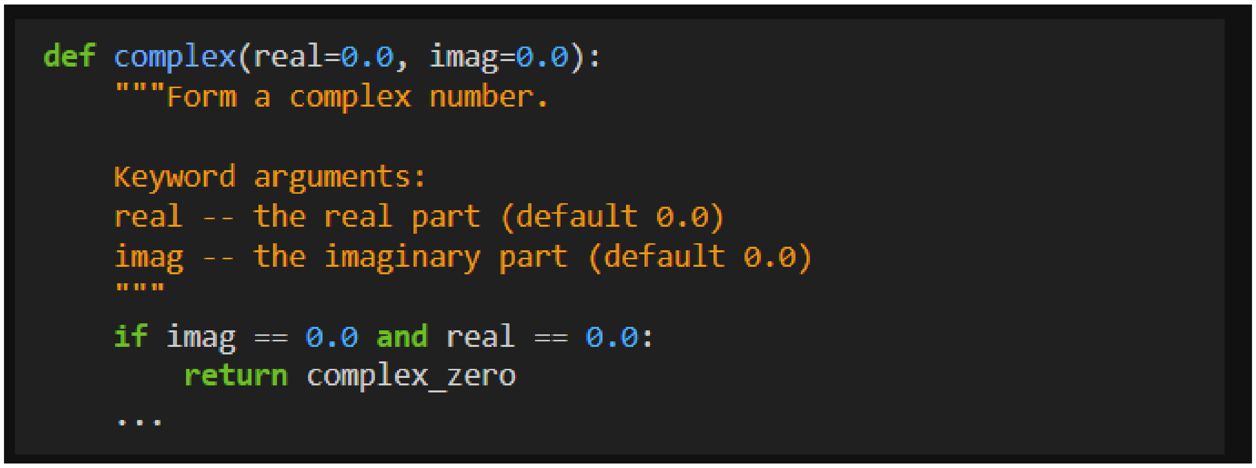
- Triple quotes are used even though the string fits on one line to make it easy to expand later.
- The closing quotes are on the same line as the opening quotes to look better for one-liners
- There is no blank line either before or after the docstring
- The one-line docstring should not be a “signature” reiterating the function/method parameters (which can be obtained by introspection). Don’t do:



```
def function(a, b):
    """function(a, b) -> list"""
```

Multi-Line Docstrings

Example of Multi-Line Docstrings:



```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Point	Description
Structure	Multi-line docstrings consist of a summary line followed by a blank line and then a more detailed description.
Summary Line	The summary line, like in one-line docstrings, should fit one line and is separated from the rest of the docstring by a blank line. It may be on the same line as the opening quotes or on the next line.
Indentation	The entire docstring is indented the same as the quotes at its first line.
Classes	A blank line is inserted after the docstring and before the class definition. Methods within the class should be separated by a single blank line.
Scripts	The docstring of a script should serve as its “usage” message, detailing its function, command line syntax, environment variables, and files
Modules	The docstring of a script should serve as its “usage” message, detailing its function, command line syntax, environment variables, and files.
Functions/Methods	The docstring for a function or method should summarize its behaviour, document its arguments, return values, side effects, exceptions raised, and restrictions on when it can be called. Optional arguments should be indicated.
Class Docstrings	The class’s behaviour should be summarized, and the public methods and instance variables should be listed. If intended for subclassing, the additional interface is listed for subclasses.

Subclassing	If a class subclasses another and its behaviour differs, it should be mentioned in the docstring using “override” or “extend” to describe the relationship with superclass methods.
Argument Documentation	Avoid using uppercase for argument names in running text and list each argument on a separate line within the docstring.
Closing quotes	If the entire docstring does not fit on a line, the closing quotes should be placed on a line by themselves to facilitate Emacs’ fill-paragraph command. *Emacs’ fill-paragraph command is a feature that automatically formats paragraphs of text to fit within a specified column width. It wraps test within the current paragraph to the desired width, adjusting line breaks and spacing as necessary to maintain readability

3.3.2 Block Comments

Block comments shall be applied to some (or all) code that follows them and are indented to the same level as that code. Each line of a block comment starts with a “#” and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single “#”.

3.3.3 Inline Comments

When the comment is on the same line as a statement, the comment should be separated by at least two spaces from the statement. They should start with a # and a single space.

3.4 Imports and Packages

3.4.1 Imports

Imports should usually be on separate lines and put at the top of the file, just after any module comments and docstrings, and before module global and constants.

Imports should be grouped in the following order:

1. Standard library imports
2. Related third party imports
3. Local application/library specific imports

A blank line should be put between each group of imports.

3.4.2 Package

Python packages should have short, all-lowercase names, although the use of underscores is discouraged.

3.5 Error Handling

3.5.1 Use Specific Exception Handling

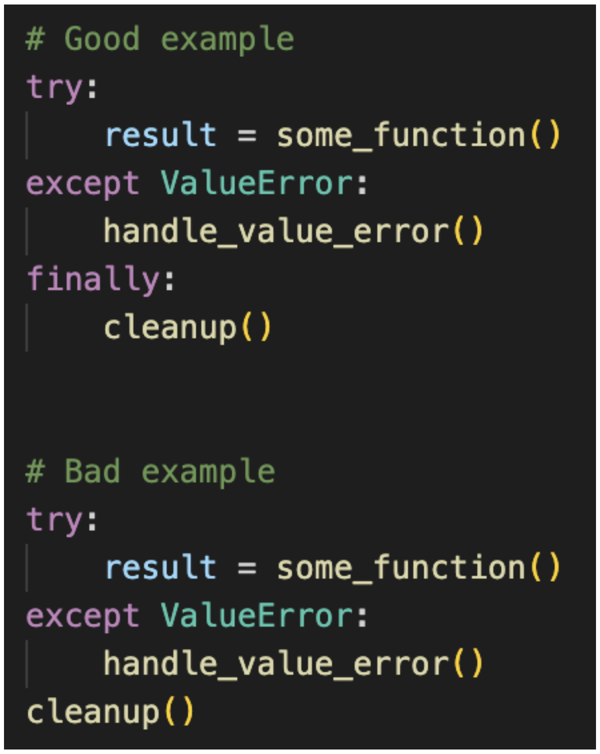
Catch specific exceptions rather than broad ones whenever possible. This helps to accurately identify and handle errors without masking other potential issues.

3.5.2 Provide Informative Error Messages

Error messages should be clear and informative. This is to aid in debugging and troubleshooting. Include relevant details such as the nature of the error and, if possible, suggestions for resolution.

3.5.3 Cleanup with finally

Use ‘finally’ blocks for cleanup operations that need to be executed regardless of whether an exception occurred.



```python
# Good example
try:
    result = some_function()
except ValueError:
    handle_value_error()
finally:
    cleanup()


# Bad example
try:
    result = some_function()
except ValueError:
    handle_value_error()
cleanup()
```