Lec2_pep

October 8, 2023

1 Good programming style: PEP-8 coding standards

- Writing clean and consistent code is good practice.
- It is important so that *others* can read and understand your code. It is of utmost importance when you work in a coding team.
- Be aware that *others* can also be you after a few weeks.
- You will be assessed on the quality of your coding.
- We will follow the PEP-8 recommendations. (Document 8 of the Python Enhancement Proposals - Style Guide for Python code).
- Link: PEP-8. Have a look and take on board recommendations relvant for your coding.

1.0.1 Code layout

White spaces

- White spaces can make a code more readable.
- But do not overdo it.

One white space after commas

[1]:
```
# do this
a = [1, 2, 3]

# not this
a = [ 1 , 2 , 3]
# and not this
a = [1,2,3]
```

One white space to follow a colon, none to proceed it.

[ ]:
```
# do this
if condition: do_something()

# not this
if condition : do_something()

# exception: slicing
my_sub_list = my_list[2:5]
```

Surround assignment = with one space on each side.

[ ]:
```
# do this
a = b+c

# not this
a=b+c
a =b+c
```

The same rule applies to comparison operators: `<, >, <=, >=, ==, !=`

[ ]:
```
# do this
if a+b**3 < a*b:
    do something
```

One or no white space before and after binary operators (‘+, -, *, /, **).

This can be used to make the priority of calculations visible,

[ ]:
```
# padding of binary operators
# do this
z = x * y

# or this
z = x*y

# not this
z = x *y

# structuring of formula putting no white space around highest priority operator
z = x * y**2
s = 3.0 + a*t**2
```

No white spaces around = signs for keyword arguments

[ ]:
```
result = my_function(x, y, one_arg=324, another_arg="Title")
```

Indentations Python uses indentations to define blocks of code, e.g., - in a loop - following a conditional statement (if ... elif ... else blocks) - function definitions

The PEP-8 recommendation is to use **spaces** not **tabs**. Four spaces per indent level.

Example

`for i in iterator:`

`....print(i)` # every dot represents one space

`....for j in another_iterator:`

`........print(j)`

By default in `Spyder` four spaces are inserted when the `tab` key is hit.

- Note that you **can not** mix tabs and spaces in Python code

- If you start working on code given to you using tabs continue doing so.

Continuation lines

- Often you will have very long lines of code. This is hard to read. PEP-8 recommends to limit the line length to 79 characters (shown in spyder).
- Use continuation lines for longer lines.

[ ]:
```
result = some_function(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9,␣
↪arg10, arg11, arg12)
```

We can split these over multiple lines like

[ ]:
```
result = some_function(arg1, arg2, arg3, arg4, arg5, arg6,
                       arg7, arg8, arg9, arg10, arg11, arg12)
```

Note the second line is vertially aligned with the character following the opening bracket.

- Python will treat lines below an open bracket and quote as continuation lines.
- Often convenient, but can also lead to hard to debug errors some lines down the code. *If you get a mysterious error message look above for open brackets or quotes.*
- For code in general one can use a backslash (\) to continue a line.

Another approach is hanging indentations

[ ]:
```
result = some_function(
    arg1, arg2, arg3, arg4, arg5, arg6,
    arg7, arg8, arg9, arg10, arg11, arg12)
```

- Here the indent is just four spaces.
- Care must be taken in function definitions because it can cause confusion. Example

[ ]:
```
result = some_function(
    arg1, arg2, arg3, arg4, arg5, arg6,
    arg7, arg8, arg9, arg10, arg11, arg12)
    return arg1+arg2+arg3
```

Solution: add another indent level and/or insert an empty line.

[ ]:
```
result = some_function(
        arg1, arg2, arg3, arg4, arg5, arg6,
        arg7, arg8, arg9, arg10, arg11, arg12)
    return arg1+arg2+arg3
```

Two good solutions if you need continuation lines for logical test for `if` statements.

[ ]:
```
# extra indent
if very_long_expression_for_condition1 and      \
        very_long_expression_for_condition2:
    do_this()
```

```
if very_long_expression_for_condition1 and     \
    very_long_expression_for_condition2:
    # A comment or an empty line to break up the code block
    do_this()
```

Empty lines are also a good way of structuring code.

If you want to put the closing bracket on its own line - align with the first non-space character on previous line - or align with the first character of the first line

There are many ways to indent continuations lines. We leave most of this to your taste. Only two rules - continuation lines need to be indented - differentiate continuation lines from other parts of the program (as above)

Line breaks of binary operators should be *before* the operator

[ ]:
```
some_result = (a       # brackets are used to make continuation lines
               + b
               + c
               + d)

# alternative
some_result = a \
               + b \
               + c \
               + d
```

Make good use of empty lines, but do not overdo it. - They can be used to indicate coherent sections (e.g. carrying out one task) - Two empty lines before and after definitions of functions and classes (more on classes later this semester)

Import of modules Modules are used to import added functionallity into a program and its namespace. They are made available through import lines *at the start of the program*. Often it is best practice to use the dot notation.

[ ]:
```
import numpy    # powerful mathematics module

x = numpy.pi / 2.0
y = numpy.sin(x)   # reminder trigonometric function use radians
print (x, y)
```

Module names can be modified

[ ]:
```
import numpy as np

x = np.pi / 2.0
y = np.sin(x)
print (x, y)
```

An alternative is to import specific functions and constants from a module

[ ]:
```
from numpy import sin, cos, tan, pi

x = pi / 2.0
y = sin(x)
print (x, y)
```

A wildcard `from` import is a **no-no**. It makes programs very confusing because it is not clear what names are in the namespace. An import from one module can overwrite names from a previous import.

[ ]:
```
# DON'T do this at home
from numpy import *
```

Do one import per line

[ ]:
```
import numpy as np
import os   # a module to make commands of the operating system available
```

Group module imports together separated by blanks.Order them 1. Standard libraries (coming with anaconda) 2. Third party libraries/local libraries/your own custom made modules

[ ]:
```
import os   # a module to interact with the operating system
import sys  # another module for interacting

import numpy as np

import my_class as mcl
```

Naming conventions

- How you chose names for variables and functions is important.
- Good variable names contribute to a well commented program
- Better use `velocity` not `v` and definitely not `a, b, c, d, e, .....x, y, z`
- A variety of styles is acceptable. Important: *consistency.*
- `lowercasewithoutunderscores`
- `lowercase_with_underscores`. (My preferred one). Note that this can be mixed with simple names without underscores).
- `UPPERCASEWITHOUTUNDERSCORES` (beloved by Fortran programmers)
- `UPPERCASE_WITH_UNDERSCORES`. (can also be mixed with simple names)
- `CamelCase` (because of the bumpy nature of the names)
- `mixedCase` (note initial lower case)
- `Capitalised_With_Underscores`

Really bad: combine use of upper case and lower case of the same name, e.g. `ABC` and `abc`. These are *different* names for a *Python interpreter* but often not for *humans*

Special conventions: - function names should be lowercase even if you use uppercase etc. - module names should be short and lowercase - class names should be CamelCase.

Comments

- Comments are an important part of every program.
- Start a a comment with a hash tag

Three types of comments: 1. Inline comments 2. Block comments 3. Doc strings

[ ]:
```
print("Hello World!")     # An inline comment
```

Inline comments can clutter the code if used too extensively. Make sure they are well separated from the code.

If more/longer comments are needed use block comments

[ ]:
```
for i in range(20):
    # This is a block comment.
    # Notice how it is at the same level of
    # indentation as the code it refers to.
    print(i)
```

Alternative you can use a docstring for a block comments - note that this is not its main purpose.

[ ]:
```
for i in range(20):
    """ This is a block commen.
    Notice how it is at the same level of
    indentation as the code it refers to.
    """
    print(i)
```

Docstrings are the comment of choice at the start of a function or a module. Every function should have a docstring at their start.

[ ]:
```
def print_my_name():
    """ This function simply
    prints my name
    """

    print ("Ralf")
```

This docstring can be accessed by using the `help` function.

[ ]:
```
help(print_my_name)
```

[ ]:
```
import numpy as np

help (np.sin)
```

What should a good docstring contain?

- Information on the purpose of the function. The method used if appropriate.
- Arguments of the function. Information on the type expected if required.
- Values returned
- Refer to PEP-257 "Docstring Conventions" for more guidance.

[ ]: