Programming Style Guidelines for Python Programs

Instructor Ann Ford Tyson
Computer Science Department, Florida State University

The following document is intended to *summarize* major style guidelines, not to explain them fully or to describe all potential situations. An entire book would be needed to do that; many have been written on the topic of what constitutes excellent programming style and design. The course graders will use this as one reference when grading class projects. If some points are not clear to you, please ask your instructors or check in the textbook for further information.

Be sure to attend all lectures and recitations, and carefully read all materials provided for this course, as style issues are frequently discussed in those venues.

FUNDAMENTAL CONCEPT TO KEEP IN MIND

Program *source code* is intended for people to READ, EASILY UNDERSTAND and WORK WITH. Think of it as a high-quality English essay which other people will read and will have to completely comprehend.

Program *object code, or machine language* is intended for computers to execute. That is an entirely different animal.

GENERAL PRINCIPLES

(1) All identifiers must be meaningful and not overly abbreviated.

variables or constants – typically use nouns, such as
```
electric_bill (not eb or ebll)
NUM_STUDENTS (not N)
```

function names - use descriptive verbs, nouns, or adjectives
```
calculate_mean
print_heading
```

(2) Functions generally should not exceed one printed page in length.

(3) Create numeric values which remain constant throughout a program as named (symbolic) constants, at the top of the file.

(4) Every control structure ideally has exactly one entrance and one exit. Most functions should have exactly one return statement.

(5) Nesting of any combination of if, for, while and so on should typically be no more than 3 or 4 levels deep.

(6) A function should only access variables

    a. created inside the function itself
    b. passed via its parameter list

    Global variables are NOT used. Global constants are often accessed directly if they have an important, consistent meaning throughout an entire program.

(7) Readability and clarity of structure are more important than efficiency in either storage or processing time. However, significant and unnecessary inefficiency is not acceptable, such as code which is repeated in a program for no valid reason.

(8) Functions can be "too short". A one or two line function which is called from one or two locations is usually inefficient. However if such a function is called from many locations, it may be acceptable. When asking yourself whether to code a task as directly executable code or as a function, consider which method will make the overall program easier to read, understand and modify later.

(9) A multi-function program must have a structure which allocates tasks among the various functions in a logical and well-designed manner, utilizing the principles of top-down design and structured programming. Each function should be functionally cohesive, i.e. perform one task or a group of very tightly related tasks.

COMMENTS a.k.a. INTERNAL DOCUMENTATION

Note that comments in a professional programming context must also include *external* documentation, a later, advanced topic area.

(1) Every program must include a main header comment with this basic format:

```
# PROGRAM <provide a meaningful program name here>
#
# AUTHOR: <your name>
# FSU ID: <your FSU user ID here, e.g. abc123>
# RECITATION SECTION NUMBER: <your section number here>
# TA NAME: <your recitation instructor's name>
# COURSE INFORMATION: <example: CGS 2930>
# PROJECT NUMBER: <number>
# DUE DATE: <give date project is due>
#
# SUMMARY
#
# A brief description of what the program does.
# Usually one paragraph or so in length.
#
# INPUT
#
# Here describe the input format. Is it interactive
# and/or from a data file? If from a file, what is the
# format of each line? How many data lines can be
# processed? Etc.
#
# BAD DATA CHECKING: If your program checks for data
#   validity, describe here what is checked for and
#   how errors are handled.
#
# OUTPUT
#
# Here describe the output values and their format.
# If tables or other complex output is produced,
# show an example.
#
# DATA STRUCTURES
#
# If any major data structures are used (lists,
# strings, tuples, sets, dictionaries, etc.) briefly
# describe them here and their role in the program.
#
# ASSUMPTIONS
#
# If any important assumptions are made, state them
# here. These will include assumptions described in
# the project write-up.
```

(2) Every function definition must have its own header comment.

```
def process_stuff():

# Here describe the input used by the function, the
# output produced, and the basic structure or algorithm
# used to get there. If a complex algorithm is used,
# describe it here. If a standard algorithm is used,
# such as a binary search or selection sort, state that
# here. This header is usually a short paragraph.
```

(3) Comment every identifier declared everywhere.

Eg. note comments are "lined up" using indentation:

```
NUM_STUDENTS = 50      # number of current students

mean_exam_score = 0    # mean score of all students
```

Eg. In functions, comment both items given in the header and items given locally

```
def find_larger(num1,  # receives 1st value
                num2): # receives 2nd value
```

(4) Other comments - various kinds of comments must be used within the code to clarify and describe what is going on. Be sure you describe what every control structure (such as a loop) is doing.

Eg. Block comments: short

```
    # This loop reads and sums the scores.
    sum = 0
    for count in range(1, 10):
        value = int(input("Enter value: "))
        sum = sum + value
```

Eg. Inline (aka "sidebar") comments

```
    if income < 0:  # handler for bad input data
```

Eg. Block comments: longer, multi-line, more extensive

```
# ------------------------------------------------- #
#                                                   #
#  The following section of code does this, that,   #
#  and these, using the Really Tricky Algorithm of  #
#  the famous mathematician, Mr. Wizard.            #
#                                                   #
# ------------------------------------------------- #
```

(5) A reader of your program should be able to read the main

header comment and the main executable section and develop a very clear idea of what the program is doing and how it does it, without having to read all the sub-functions.

FORMATTING PROGRAM SOURCE CODE

(1) Each statement should begin on a new line.

(2) Appropriate indentation is used to improve readability and to provide correct syntax in all code. The unit of indentation is 4 spaces. Do not mix the use of spaces and tabs. Spaces are preferred in newly created code.

(3) The source code and comments on each line must appear in the first 80 or fewer columns of the line. The program, when printed, must fit within the margins of a standard 8 1/2" by 11" printer sheet, printed in portrait mode.

(4) At least one blank line must appear before major code sections.

(5) Comments on a line of code which are continued on successive lines are indented so that all lines are "lined up" under the initial comment.

(6) At least 1 blank space must appear before and after '=', and the relational operators (==, >, <, <= etc). At least 1 space must appear after '#' in a comment. Arithmetic, logical and other binary operators should have 1 space both before and after. Spacing can be used to clarify many equations. At least 1 space should be used after ',' and ':'. Parentheses can be used to clarify code.

```
Eg.      result = x + y              # this is acceptable

         result = x*y + w*z          # not very good

         result = (x * y) + (w * z)  # this is more clear
```

Compare the following 2 statements for readability:

```
         flag=(x<y)and(not(y<z))      # not good

         flag = (x < y) and (not(y < z)) # better
```

(7) The use of additional blank lines and blank spaces is recommended whenever such use improves readability. Avoid an excessive use of blanks, which can impair readability.

(8) Functions are separated from one another by using at least two blank lines. It is also ok to use rows of asterisks or some other delimiter to indicate a new function. For example, after the series of blank lines, you might do something like this:

```
# ********************************************** #
def print_headings ():
  etc.
```

You can also be quite fancy if you like and draw blocks of characters around function headings, etc.

(9) If a statement must be split across lines, split it at a logical position, and align successive lines. You can use the line continuation marker '\' or enclosed items in parentheses or braces where appropriate.

INDENTATION

See the textbook and programs presented live in class for many examples of general indentation guidelines and indentation when using braces. Indent 4 spaces when called for. Note that Python uses indentation to correctly parse your code, so incorrect indentation is typically a syntax error. Do not mix indentation with SPACES and TABS as this will almost always cause syntax problems.

Examples:

```
  while expression:
     statement1
     statement2
     statement3
```

or

```
  if expression:
     statement1
     statement2
  else:
     statement1
     statement2
```

USE OF MIXED CASE

A mixture of upper and lower case *must* be used to maximize readability. Remember that Python is case-sensitive, so you must use the same cases in every instance of a given identifier. The following are common style conventions in current use.

Variable names start with lowercase and are typed in lowercase with underscores to separate words or other meaningful parts, as in

```
  client_name = "Bill Gates"
```

Named constants are typed using ALL uppercase using the underscore as a separator, as in

```
MAX_NUM_ITEMS = 1000
```

Notes on separators: For variable names and similar identifiers, be *consistent* and use either *camel case or mixed case*, as in

```
NumberOfCreatures = 10   # camel case

numItemsSoldToday = 55   # mixed case
```

or *underscores*, as in

```
number_of_creatures;
```

to make your identifiers more readable.

Consistency is typically more important than which style is chosen. *LOCAL* style rules must always be followed. Local means the rules in the course you are currently taking, or the software development company or environment you currently work in. If you are working with pre-existing code, it is often most important to be consistent with the pre-existing style.

OUTPUT CONTENTS AND FORMATTING

(1) The beginning of a program's output must be labeled clearly with an appropriate overall title. This is the *opening* message.

(2) All numeric values must be formatted to maximize readability. For example, monetary values would be printed to 2 digits of precision.

(3) Output must always "stand on its own", i.e. titles, labels and meaningful messages must make it obvious to the reader what the output means.

(4) White space (blank lines and blank spaces) must be used to separate sections of output to maximize readability.

(5) Tables must have meaningful titles, row and column labels as needed. All table data should be lined up neatly.

(6) Any form of output, even if not specifically mentioned here, must be well-formatted and clear. Output, when printed, must fit within the margins of a standard 8 1/2" by 11" printer sheet, printed in portrait mode. Output must fit neatly within the page and, at most, an 80-column output window. No wrapping.

(7) All input data must be echoprinted. Note that if you are using interactive input in the text window, this is accomplished automatically as you enter values. If you read in data from a file, you will have to explicity print input values from your code.

(8) When a program requires interactive input, *friendly* and *informative* prompts must be printed for the user. By friendly we mean pleasant and helpful. By informative, we mean that the prompts need to tell the user what he/she needs to do in a very clear way. Prompts should also be well-formatted, like any other form of output.

(9) The end of a program's output must be displayed clearly using an appropriate *closing* message.

FINAL NOTES

See the Python Style Guide PEP 8 online for further information. Effort has been made to make this document as compatible as possible with PEP 8, however while being most appropriate for *introductory* programming courses.

The PEP 8 can be found at http://www.python.org/dev/peps/pep-0008/

Author: Ann Ford Tyson
Last Update: July 2019