Python Code Style Guide

1. Introduction

Most of this course focuses on the elements of a programming language and how to employ them to write correct code. But, like human languages, progrmming languages also leave considerable room for *style* (how we choose to express a particular implementation).
One might reasonably assume that stylistic decisions are entirely matters of personal choice.

In 1959 Strunk and White's "The Elements of Style" asserted that the primary purpose of written language was communication, and that stylistic elements can be evaluated based on the extent to which they facilitate understanding; Remove what is superfluous, and ensure that each word and sentence communicates its meaning clearly.

Similar principles apply to thoughts expressed in a programming language. Different programming communities may observe different stylistic conventions, but most communities do have guidelines to improve the readability of their code. Some of these rules are to reduce the likeihood of common errors. Others merely establish common ideoms whose uses will be immediately understood by experienced readers.

For these reasons, it is valuable for you to complement your understanding of syntax and sematics with an appreciation for the elements of programming style:

- what are the key stylistic elements, and how do they effect code readability?
- examples of common usage conventions, and what advantages do they confer?

2. Format and use of White Space

The simplest stylistic elements of a written program involve the ways in which the words/tokens are arranged into lines. Newlines, tabs, and spaces are collectively referred to as "white space". We see white-space conventions in written prose:

- the flow of sentences within a paragraph shows them to be connected thoughts.
- the blank lines between paragraphs show them to be distinct thoughts.
- the sub-indentation of a bulleted list shows an enumeration of distinct sub-elements.
- a new chapter or sub-section may start on a new page, to clearly indicate the start of a new subject.

These are examples of how the arrangement of the characters on a page can suggest (or reinforce) the semantic structure of the presentation, making it easier for the reader to absorb the presented material.

White space can be employed very similarly in most programming languages.

2.1 Indentation

Indentation is commonly used to indicate subordination:

- the methods within a class are indented to the right of the class definition.
- the code within a method is indented to the right of the method definition.
- in an **if** statement, the **if** and **else** clauses are indented to the right of the **if** and **else** keywords.
- the body of a loop is indented to the right of the **while**/**for** statement that controls the loop.

In most languages this subordination is defined by matching curly braces (or parentheses, or square or angle brackets) ... and indentation is a purely stylistic element to make this structure more obvious. Python does not use braces/brackets; The hierarchical structure of the code is expressed entirely through indentation. Thus, correct indentation in Python is not merely a matter of style.

Should we achieve our indentation with tabs or spaces? Arguments can be made for either convention, but there seems to be a general consensus that mixing tabs and spaces is a bad idea (because the per tab indentation may be different on different systems). The Python PEP-8 standard recommends use of 4-spaces per level of indentation (rather than tabs). This enables more levels of nesting within a limited line length.

2.2 Vertical White Space

Just as inter-paragraph spacing is used to separate distinct thoughts, programming styles encourage the use of vertical white space to separate logically distinct code elements: - classes or methods within a module - variable declarations/initializations from code - distinct processing steps within a method

The Python PEP-8 standard recommends: - a single blank line between distinct blocks of code within a method - a single blank line between methods (or declarations/initializations) within a class - two blank lines between methods (or declarations/initiaizations) that are global (or not within a class).

How do we decide whether or not the next line of code should be separated from the statement before it? This is a case-by-case trade-off:

- adding blank lines between distinct steps emphasizes that a previous step has been completed and a new step is beginning.
- adding numerous blank lines to a method (or set of declarations/initializations) may make that block of code so long that it no longer fits on a single screen, making it more difficult to read and comprehend.

2.3 Line Length and continuations

In the previous century, when programs were written on `punched cards`, it was not possible to code a statement that was more than 80 columns wide. Variable width windows (and more liberal languages) have largely done away with such hard limitations. But it is still the case that human beings can be easily confused when trying to read expressions that are too wide or fold across multiple lines. For this reason, the Python PEP-8 standard still suggests a maximum line width of 79 characters.

If you find that a line is becoming very wide, you should give some thought to how the line can best be continued across multiple lines. Ideally, the break-up of a long line can be done in a way that enhances its readability:

- if a method call passes a great many long parameters, we could put each parameter on its own line (rather than breaking a single argument across a continuation line).

```
mymethod(first_parameter, second_parameter,
        third_parameter, fourth_parameter,
        fifth_parameter, sixth_parameter)
```

- if a long expression is a sum of many terms, we could put the line break between terms (rather than breaking a single term across a continuation line).

```
root1 = (-b + sqrt((b**2) -
        (4*a*c))) / (2*a)
```

Because the above parameter list and expression are parenthesized, Python will know that it has not yet seen the closing paren, and that the call or expression is being continued onto the next line. If the statement cannot be broken within a parenthesized expression, a backslash (') at the end of a line tells Python that the statement will be continued on the next line.

```
if first_condition and second_condition and third_condition \
   and fourth condition and fifth condition:
    do something
```

But, do not combine multiple statements (e.g. an **if** and the if-clause, or **while** and loop) on the same line.

2.4 Horizontal White Space

For vertical white space we observed that there is a tradeoff between the advantages of extra space to delimit distinct elements and the disadvantages of making a block of code too long to fit on a single screen. Similar arguments apply to the use of horizontal white space:

- while the language uses commas and parentheses to infer the structure of a complex expression, inserting blanks to separate distinct terms may be

much easier for a human being to perceive.
- adding too many spaces to an expression may make it too long to fit on a single line or introduce an unfamiliar variation to an otherwise well-known ideom.

Stylistic guidelines may include suggestions to eliminate gratuitous horizontal white space:

- no space between the name of a method and the left paren that introduces its parameters.

```
my_method (first_arg, second_arg)
```

- no space between a pair of enclosing parentheses/braces and the first or last term within them.

```
my_method( first_arg, second_arg third_arg )
```

- no space between a parameter and the comma that introduces the next one

```
my_method(first_arg , second_arg , third_arg)
```

- consistant space (i.e. none or one) between binary or assignment operators,
- always use spaces around relational operators.

```
if x <= 4:
    ...
```

- no spaces around the assignment operator for keyword parameters

```
my_method(radius=3.0, color="green")
```

- no trailing (at the end of a line) whitespace

3. Comments

Programming languages are intended for expressing instructions to computers. They are not designed for explaining concepts to human beings. As such the design of a program (what it is supposed to do and how it achieves those goals) is not always obvious from a reading of the code. Most programming languages allow comments (explanatory notes) to be added to the code.

3.1 Module Descriptions

Before we can start reading the code in a new file (or moodule), it is useful to understand:

- what the purpose of that module is within the larger program.
- how the code within the module is organized.

These are somewhat analogous to the introduction and the table of contents in a book. This information is generally placed in a block comment at the very

top of each file or module. In Python such comments are called DocStrings, and begin/end with tripple quotes. One DocString is regarding the method description, which will be detailed in next section (3.2); the other one is regarding the submitter identification at the beginning of a module in the python file that you submitted. The submitter identification should include:

- Course, assignment name
- The author information
- a standard Module docstring
  - summarizing the purpose of this module
  - summary of functions, classes, and exceptions exported by the module

A very simple example might be:

```
"""
Assignment #9 - a new class to represent Chess boards

Author: Your name, Your lab section

Note: the main() method creates a Chess board and
      runs a set of operations to exercise all of
      the methods.
"""
```

3.2 Method Descriptions

Anybody who is working with a module needs to understand, for each method:

- what the method does
- descriptions of each of its parameters
- description of its return value

This information is generally placed in a block comment at the top of each method. In Python these DocStrings appear immediately after the method declaration. A simple example might be:

```
def num_students(department, course_id, section, wait_list):
""" count number of students in a course/section

    :param department (str): name of department
    :param course_id (int): number of the course
    :param section (int): section number
    :param wait_list (boolean): include wait-listed students?

    :returns: (int) number of students

    :raises LookupError: unable to find specified course or section
"""
```

3.3 Algorithmic Descriptions

Much code is so simple and clear that it is obvious what it is doing and why it is doing it. In such situations, descriptive comments might actually be clutter and make the code more difficult to understand. If the code is obvious, let it speak for itself.

If a method implements a multi-step process, its readability may be enhanced by enumerating the steps in the method's DocString, or by putting a one-line description of each step immediately in front of the code that performs it.

The following example is regarding processing expense of a credit card:

```
def process_expense(cc_number, amount):
"""
Process expense and update balance for credit card users

:param int cc_number: a 13-digit credit card number
:param float amount: the amount of the transaction
"""
    # Step 1: verify credit card number, verify(cc_number) returns boolean
  if verify(cc_number):
    # Step 2: check balance of the account, check_balance returns boolean
      if check_balance(cc_number, amount):
      # Step 3: bill amount to the balance of account cc_number
        bill_balance(cc_number, amount)
    else:
        print("It is not approved because of exceeding balance limit")
  else:
      print("It is not a valid credit card.")
```

In some cases the algorithm being implemented may be very complex, or the code may be predicated on non-obvious assumptions. In such cases explanations should be placed in a block comment at the top of the non-obvious code.

For example, we use bubble sort to sort an array of integer nunmbers in acsending order.

```
n = len(array)

# traverse all the elements in array
# compare the element with its next one
# swap if the element is greater than next
for i in range(n):
  for j in range(n-i-1):
    if array[j] > array[j+1]:
      array[j], array[j+1] = array[j+1], array[j]
```

3.4 Checking your DocStrings

Within the Python interpreter, you can use the help() function to look at the DocStrings for a module or method: * help(module_name) * help(module_name.method_name)

If you are working from the command line, you may be able to run the pydoc command on your module to get a complete list of all of the the module and method DocStrings.

4. Variables, Methods, Classes and their names

Well chosen names greatly contribute to the understandability of code.

4.1 Mnemonic names

The name of a variable should suggest its meaning. Consider the following code (from the above num_students method):

```
if (wait_list):
    return enrolled_students + waitlisted_students
else:
    return enrolled_students
```

The chosen variable names make it fairly obvious what the code is doing.

Simiarly, the name of a method should suggest what it does, and the names of the parameters should suggest the meaning of each. Again, from the above example:

```
    def num_students(department, course_id, section, wait_list):
```

Even without the DocStrings, the purpose of the method and the meanings of its parameters are fairly clear. Similarly, the name of a class should suggest the abstract entity that the class represents.

4.2 Case conventions

Coding style conventions often specify different types of names for different things. In Python, for instance, the PEP-8 standard recommends:

- class names should use `CamelCaseWithInitialCapitalLetter`
- public method and variable names should use `lower_case_with_underscore_between_words`
- private methods and variables should use `_lower_case_with_leading_underscore`
- constants should use `UPPER_CASE_WITH_UNDERSCORE_BETWEEN_WORDS`

Such rules may not be enforced by the language, but they are likely to be strictly enforced within a particular programming community.

The advantage of such conventions is that we can tell what kind of thing a name refers to based solely on its form, without even having to look up its DocStrings.

4.3 Constants

There are many reasons to include constants (literal numeric values or strings) in a program. Examples might include:

- the maximum allowable length of a string or list
- commonly used values (e.g. `PI=3.14159`)
- a string that has special meaning (e.g. a key-word in the data)
- numerical values with special meaning (e.g. `LEFT=1`, `RIGHT=2`)
- a limit on how many times a process should be repeated

It is generally considered poor form to put literal strings or values in our code. Rather it is common practice to assign those values to a named constant, and to use that named constant in the code:

- a numeric value is not mnemonic, whereas a named constant can suggest the meaning of the value.
- if a value is used in multiple places, there is a danger that different values will be used in different places. If all of those references are to a single named constant, all can be assured of using the same value.
- some of these values may change over time (e.g. because we decide to support longer strings or lists). If the size is referenced as a named constant, we only have to change the value assigned to that constant, and every use of that constant throughout our program will automatically be updated to use the new value.

4.4 Method parameters and return values

In addition to being reasonably named, the parameters to a method should:

- be information in a form (e.g. units) that the caller is likely to have or be easily able to compute.
- be readily understandable by a caller who does not understand how the method will be implemented
- enable a reasonable implementation of the specified functionality.

Similarly, the returned value should be something that is easily understood and used by the caller.

5. Code

Thus far we have talked about things (like formatting, comments, and names) that do not affect the meaning of the code. But, there are also subtle aspects of coding where beginners often have problems, and a better understanding of the language enables clearer, more concise, and even more correct expressions of our intentions.

5.1 Expressions

Use of intermediate variables

If an arithmetic expression is particularly long or complicated it can often be simplified by computing values for individual terms, storing them in (well named) variables, and then compute the final value as an expression in terms of those variables.

```
delta_x = x2 - x1
delta_y = y2 - y1
distance = sqrt(delta_x**2 + delta_y**2)
```

Boolean Expressions

In terms of boolean expression, you should NOT have anything like:

```
    if boolean_expression == True:
```

or

```
    if boolean_expression == False:
```

instead use:

```
    if boolean_expression:
```

or

```
    if not (boolean_expression): # or some other way of negating the expression
```

If `function(x, y)` returns a boolean value, you should NOT have anything like:

```
    if function(x, y):
        return True
    else:
        return False
```

instead use:

```
    return function(x, y)
```

Sometimes people use nested if statements to compute a result that is based on one or more conditions. But this can often be done more simply and clearly with a boolean expression.

For example:

```
if (class_open and num_enrolled <= capacity):
    return True
else:
    return False
```

could be much more simply written as

```
return class_open and num_enrolled <= capacity
```

Use of return values

If you want to make a computation or decision based on the value returned from a call to some function, you have a choice:

- put the function call in the expression, and the expression will be evaluated based on the returned value. This is the preferred option if the returned value will only be used once:

```
if (bonus + random.randint(1,20) >= armor_class):
    hit()
```

- store the returned value in a variable, and then use that variable in the expression to be evaluated. This makes sense if the same return value is to be used in multiple tests:

```
roll = random.randint(1,6) + random.randint(1,6)
if (roll == 7 or roll == 11):
    win()
elif (roll == 2 or roll == 3 or roll == 12):
    lose()
```

5.2 Efficiency

Failure to recompute a value

Sometimes programmers make simple mistakes like: compute a value and store it in a variable, but fail to recompute it when one of its tributary input values changes. When code uses that (now stale) value, an incorrect decision may be made.

For example, given a list of integer numbers num_list, find out the number that is closest to the average of these numbers. The following code seems working:

```
def average(alist):
  """
  Get the average of the input list of integer numbers

  :param list alist: a list of integer numbers
  :return float: the average of numbers in the list
  """
  total = 0
  count = 0
  for i in alist:
      total += i
      count += 1

  return total / count
```

```
def closest_to_average(num_list):
  """
  find the number that is closest to the average of numbers in the list

  :param list num_list: a list of integer numbers
  :return int: the number that is closest to the average
  """
  min_diff = INFINITY # INFINITY is a very large number

  for n in num_list:
      if abs(n - average(num_list)) < min_diff:
          closest = n

  return closest
```

However, once a closer number is found, the min_diff is not updated, which will lead to an incorrect result. So, the code should be:

```
def average(alist):
    ......


def closest_to_average(num_list):
    """
    find the number that is closest to the average of numbers in the list

    :param list num_list: a list of integer numbers
    :return int: the number that is closest to the average
    """
    min_diff = INFINITY # INFINITY is a very large number

    for n in num_list:
        if abs(n - average(num_list)) < min_diff:
            closest = n
            # Update the value of min_diff to achieve the correct result
            min_diff = abs(n - average(num_list))

    return closest
```

Whenever you store a value in a variable for future reference, consider how long that value will continue to be valid and what might cause it to change.

Recompute the same value

Sometimes, programmers make the opposite mistake that the program recomputes a value (upon every iteration) inside of a loop, or upon every invocation of a method ... even though its value does not change. This is a wasted

computation that adds clutter to the code.

Just like the above example, the `average` function has been called for two times: one in the conditional expression after if, the other one inside the if clause, both of which are inside a for loop. Due to the nested loops, the total running time is O(n^2).

Actually, these two computations result in the same value. So, in order to save the computational resource and make the code more efficient, e.g., O(n) running time, the following code should be considered:

```
def average(alist):
    ......


def closest_to_average(num_list):
    """
    find the number that is closest to the average of numbers in the list

    :param list num_list: a list of integer numbers
    :return int: the number that is closest to the average
    """
    min_diff = 1000  # INFINITY is a very large number

      # calculate the average outside of the loop to achieve efficiency
    average_value = average(num_list)

    for n in num_list:
        diff = abs(n - average_value)
        if diff < min_diff:
            closest = n
            # Update the value of min_diff to achieve the correct result
            min_diff = diff

    return closest
```

Loop exiting and continuation

A simple for loop performs the same computation for every element of the specified range. A simple while loop continues to perform the same computation as long as the loop condition remains true. But some loops are more complex:

- at some point in the mid-loop computation, it may become clear that we are done. There is no need to continue iterating over the remaining range or re-checking the loop condition. Python (like many other languages) has a **break** statement, which means that we should immediately exit the current (or inner-most) loop.
- more complex loops may treat different range values (or current items)

very differently. We may reach a point where we are through processing for the current iteration/value, and all of the remaining code in the loop body is irrelevent (not applicable in this case). Python (like many other languages) has a **continue** statement, which means that we are done with this iteration of the current (or inner-most) loop and should immediately return to the next element of the range or checking the loop condition.

The use of **break** or **continue** statements to simplify code within the body of a loop is illustrtated at the end of the next section.

if/else factoring

If there are three different boolean conditions that can affect a computation, we could find ourselves writing three-level-deep if statements with eight distinct **if**/**else** clauses (for each possible combination of those three conditions). Such code would be difficult to understand, and perhaps even difficult to write correctly.

```
if (condition1):
    if (condition2):
        if (condition3):
            deal with T/T/T
        else:
            deal with T/T/F
    else:
        if (condition3):
            deal with T/F/T
        else:
            deal with T/F/F
else:
    if (condition2):
        if (condition3):
            deal with F/T/T
        else:
            deal with F/T/F
    else:
        if (condition3):
            deal with F/F/T
        else:
            deal with F/F/F
```

But perhaps the solution does not require distinct code for each of the eight possible combinations. Perhaps all cases involving condition3==False can be handled in the same way. By moving that test up to be the first one, we can eliminate half of our possibilities:

```
if (not condition3):
    common code for all cases where condition3 is False
elif (condition1):
    if (condition2):
```

```
        deal with T/T/T
    else:
        deal with T/F/T
elif (condition2):
    deal with F/T/T
else:
    deal with F/F/T
```

If the required processing is different, depending on combinations of multiple conditions, look for ways to factor out common cases to reduce the complexity of the required code.

If this decision is happening within a loop, it may be possible to use **break** or **continue** to greatly reduce the need for nested **if**s and **else**s:

```
for ...
    ...
    if (not condition3):
        common code for T/T/F, T/F/F, F/T/F and F/F/F
        continue
    if (condition1):
        deal with T/T/T and T/F/T
        continue
    if (condition2):
        deal with F/T/T
        continue
    deal with F/F/T
```

Gratuitous object creation

Programs regularly instantiate a variety of objects (e.g. a random number generator or a data plot) in order to use the services of that class. Each new object we create incurrs a cost (in both cycles and memory). So we should not create more objects than we need.

Before creating a new object, ask yourself if this is really distinct from previously created objects, or if we can simply continue to use (or reuse) a previously created object.

For example, given a class called `SalesTag` that has attributes such as `state`, etc, and functions such as `get_tax_rate`, `set_state`, etc, to calculate the total tax of a list of retail items, the following code may work:

```
def sales_tax(retail):
    """
    Get the total sales tax for a list of retail products

    :param list retail: a list of retail items, each of which has
        attributes such as price, state, etc.
    :return float total_tax: the total tax of all the items in the list
```

```
    """
    total_tax = 0

    for item in retail:
        tag = SalesTag(item.state)
        tax_rate = tag.get_tax_rate()
        total_tax += item.price * tax_rate
    return total_tax
```

Here, we created a number of tag objects due to the object creation inside the for loop. Actually, this will consume more computer resources, since most of these objects are not necessary. To avoid unnecessary objects, we can reuse an existing object as follows, to achieve more efficiency.

```
def sales_tax(retail):
    """
    Get the total sales tax for a list of retail products

    :param list retail: a list of retail items, each of which has
        attributes such as price, state, etc.
    :return float total_tax: the total tax of all the items in the list
    """
    total_tax = 0

    # The tag object will be created outside the loop and will be reused
    tag = SalesTag()

    for item in retail:
        tag.set_state(item.state)
        tax_rate = tag.get_tax_rate()
        total_tax += item.price * tax_rate
    return total_tax
```