Course Notes: Style

Portions of these notes are new for the Spring 2022 semester, and they are very much a work in progress, and they'll likely be updated/augmented/refined throughout the semester. If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch by posting to the course notes section of the forum or sending a private message to the instructors via the forum.

Table of Contents

1) Introduction

People often refer to "good style" when talking about programs/code, but, interestingly, that word can mean a lot of different things depending on context. In this document, we'll try to explain what "good style" means in the context of 6.009, as well as to provide some suggestions for incorporating elements of "good style" into your own programs.

One notion of style relates to detailed character-by-character desisions about how code is *formatted*, like how much whitespace to use when indenting, where parentheses should go, etc. The Python Software Foundation does publish/maintain an official style guide called PEP 8, and this convention is probably the most widely-used style convention for Python code, so it's not a bad idea to get into the habit of following those guidelines.

That said, while conventions related to this notion of "style" are perhaps worth knowing and caring about, we are not going to focus on them in 6.009 (you are welcome to put your parentheses wherever you like so long as your programs still work). Rather,

we'll be concerned with style from a higher-level perspective that focuses on how your code is *structured*, rather than how it is *formatted*. So rather than focusing on writing code that *looks* pretty, we'll focus on writing code that is as easy to read, write, understand, and debug as possible.

2) Guidelines

What constitutes good style and good design is, in some ways, a subjective question, and it depends somewhat on the particular program you're trying to write. However, what follows are some more-or-less general guidelines, each of which is detailed in a following section. While there may be reasons not to follow some of this advice in particular cases, they are generally good things to keep in mind when writing code.

- **Don't Repeat Yourself (DRY)**: Avoid multiple fragments of code that describe redundant logic.
- **Names Matter**: Choose concise, descriptive names for functions, parameters, and other variables.
- **Documentation Matters**: Use docstrings and comments to describe assumptions and document non-obvious features of the code.
- **Generality Wins**: Define logic, functions, and programs as generally as possible.
- **Plan for Change**: Where possible, make programs that are (relatively) easy to change, should the need arise.

Following these guidelines will help you to:

- improve the readability of your code (other people *will* read your code, even if those other people are all either 6.009 staff or future versions of yourself)
- reduce the number of errors in your code, and make it easier to spot/fix them
- make it easier to make changes to your program if you need to
- minimize the amount of code you write (the less code we have, the fewer opportunities there are for bugs!)

Note that you do not need to come up with a stylistically perfect design on your first try! Designing for style takes time, experience, and practice. When starting, it is likely a good idea to get things to work first, but to keep an eye out for opportunities to improve stylistically as you're working (and after you are "done"), and to be open to making changes that improve the style of your code. This kind of iterative approach of successively refining a program can be really productive; and with more time and experience, you will be able to "see" more of these stylistic improvements even before implementing a first solution, and your plans will start to include these ideas from the get-go.

With that in mind, let's explore some of these ideas in a bit more detail.

2.1) Don't Repeat Yourself

Modern text editors have a nice feature called copy/paste, which makes it easy to duplicate a piece of code you've already written. It may often be tempting to do this when you encounter a problem that is similar to another problem you've already solved (the voice in your head says, *Hey! I've already written something that's almost right here; let's just copy/paste the code, tweak it a little bit, and then we've solved the new problem!*). Every experienced developer has heard this Siren song before, but it's important to avoid this temptation.

The main fundamental risk that is intrinsic to duplicated code is that if you have identical code (or very similar code) in multiple places in your program, then noticing an error (or some other opportunity for improvement) in one copy requires making appropriate changes to *all* copies. Not only is this a pain, but it's really easy to forget to make the change in one or more of the copies.

You should avoid duplication of code like you would avoid crossing the street without looking. Copy/paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the chunk of code is that you're copying, the riskier it is.

Keeping an eye out for duplicated code as you're writing code (as well as after), and rearranging things to reduce the amount of code that is duplicated, can save a lot of trouble later on.

2.1.1) The Rule of Three

Not all code duplication comes from copy/paste, though. Some cases of duplicated code arise from manually typing the same (or very similar) things multiple times.

If you find yourself re-writing the same short expression over and over, that might be a good sign that you can store that in a variable as an intermediate result, or construct a loop to repeatedly compute related values. Along those same lines, if you find yourself copy/pasting a block of code to compute a result, that might be an opportunity to define a "helper" function for that behavior, and to call that function multiple times rather than duplicating the same structure multiple times.

Some people follow the "rule of three" (if a piece of code or a computation is repeated three or more times, abstract that computation into a function or a loop). That's a reasonable approach for small pieces of code, but a better approach for larger chunks might be the rule of *two*.

2.1.2) Boolean Laundering

One specific case where repeated code often shows up is so-called "Boolean laundering," which is perhaps best demonstrated by example. Consider the following piece of code (perhaps you've seen a structure like this in your own code in the past!):

```
def is_negative(x):
    if (x < 0) == True:
        return True
    else:
        return False
```

It may not look like it since each line is actually unique, but there is actually an awful lot of redundancy in this little example. Let's look a little closer and see how we could achieve the same result in a clearer and more efficient way.

Importantly, the expression `x < 0` will always evaluate to a `Boolean`, so it will either be `True` or `False`. If it is `True`, then we will enter the first block of the conditional (since `True == True` evaluates to `True`), and if not, then we will enter the second block of the conditional (since `False == True` evaluates to `False`).

If the parenthesized portions of that last sentence sounded a little silly/redundant to you, that's a good sign, as we're perhaps already seeing one place to improve the code above. Because `x < 0` is a Boolean, `(x < 0) == True` will always evaluate to exactly the same thing as `x < 0`. So we could rewrite this function as follows:

```
def is_negative(x):
    if x < 0:
        return True
    else:
        return False
```

This is progress! We've removed a bit of redundant compuatation. But it turns out that we can go quite a bit further. Remember that `x < 0` will evaluate to a Boolean, so it will either evaluate to `True` or to `False`. In the case where it evaluates to `True`, we enter the first block of the conditional and return `True`; and if not, we enter the second branch and instead return `False`. So in fact we could write the following:

```
def is_negative(x):
    return x < 0
```

2.2) Names Matter

Choosing good names for functions, parameters, and other variables will help make your code much easier to understand. Names of functions should describe what they do. Variable names should describe what they represent (not just their types!). Single letter names are okay in some situations, but use such names sparingly.

2.2.1) Concise, Descriptive Names

There is a lot of freedom in choosing variable names in the programs you write, but it is a good idea to try to balance choosing descriptive variable names with choosing short variable names that are reasonably easy to type.

Single-letter names should generally be avoided, as should names like `tmp`, `temp`, and `data`. Many variables are temporary, and technically every variable is a type of data, so those names to not mean anything. It is better to use a longer, more descriptive name, so that your code reads clearly by itself. It is also generally a good idea to avoid abbreviations, since, while their meanings may be evident to you, they may not be to someone else.

Good, descriptive variable names make it easier to understand code at a glance. For example, it takes a while to figure out what the following program does, primarily because of poorly chosen names:

```
def a(x):
     return 2 * A * x

def b(x):
    return A * x**2

def c(x):
    return 4 * b(x)

def d(x):
    return 4/3 * A * x**3
```

Its readablility could be drastically improved by choosing variable names more carefully:

```
def circle_circumference(radius):
     return 2 * PI * radius

def circle_area(radius):
    return PI * radius**2

def sphere_surface_area(radius):
    return 4 * circle_area(radius)

def sphere_volume(radius):
    return 4/3 * PI * radius**3
```

Of course, it is possible to go too far in the direction of descriptive variable names. The following code, which borders on the absurd, is maybe even more difficult to decipher than the first example:

```
def circumference_of_a_circle_given_its_radius(the_radius_of_the_shapes_for_which_we_want_to_compute_values):
     return 2 * the_ratio_between_the_circumference_and_the_diameter_of_a_circle *
the_radius_of_the_shapes_for_which_we_want_to_compute_values

def area_of_a_circle_given_its_radius(the_radius_of_the_shapes_for_which_we_want_to_compute_values):
    return the_ratio_between_the_circumference_and_the_diameter_of_a_circle *
the_radius_of_the_shapes_for_which_we_want_to_compute_values**2
```

```
def surface_area_of_a_sphere_given_its_radius(the_radius_of_the_shapes_for_which_we_want_to_compute_values):
    return 4 * area_of_a_circle_given_its_radius(the_radius_of_the_shapes_for_which_we_want_to_compute_values)

def volume_of_a_sphere_given_its_radius(the_radius_of_the_shapes_for_which_we_want_to_compute_values):
    return 4/3 * the_ratio_between_the_circumference_and_the_diameter_of_a_circle *
the_radius_of_the_shapes_for_which_we_want_to_compute_values**3
```

Interestingly, Python will happily run any of these pieces of code, and the results will be the same if we call any of the functions! But it's important to write programs not only with an eye toward correctness, but also with an eye toward clarity; one of these is far easier to understand than the others.

2.2.2) Magic Numbers

Whenever possible, you should avoid the use of hard-coded constant values, often called "Magic Numbers" because they show up out of nowhere with no explanation, as if by magic. Magic numbers can make your code more prone to bugs, as well as more difficult to read and understand.

For example, the following piece of code has a magic number in it:

```
def total_cost(raw_cost):
    return raw_cost * 1.0625
```

Here, it's really not clear what `1.0625` means. One way to clear this up would be with a comment describing the nature of that number, but, particularly if that number shows up in multiple places, a far better way is to declare that number as a named variable with a good, clear name, and to refer to it by that name instead.

In this case, making that kind of change might lead to the following code:

```
MASSACHUSETTS_TAX_RATE = 1.0625

def total_cost(raw_cost):
    return raw_cost * MASSACHUSETTS_TAX_RATE
```

This kind of a change has a few advantages over using a magic number. Among other things:

- A number is less readable than a name. It's more difficult to glance at the first version of this function and understand its purpose.
- Constants may need to change in the future. Using a named variable instead of repeating a magic number makes it easier to respond to those kinds of changes in the future.
- Constants may depend on other constants! In fact, some magic numbers are the result of computations *done by hand by a programmer* and involving other magic numbers. Code is far clearer and more adaptable if we represent these numbers as named variables, and if we compute the related constants using explicit expressions in terms of other variables.

It's perhaps worth noting, too, that if you have multiple magic numbers in your code, and those numbers are related to each other in some way, it may make sense to store them in a list or a dictionary, rather than storing each as its own variable.

After reading this section, you may ask yourself: *What about things like π or the gravitational constant, that I know will never change? Surely that's fine to hard-code, since it's definitely not going to change...*

While you are right that the value associated with π is not ever going to change, there are still at least a few reasons to prefer using a named variable to refer to π, as opposed to referring to it by value:

- As with certain other magic numbers, π is complicated to express, so repeating it multiple times may actually provide an opening for bugs to sneak into your code. For example, perhaps I put `3.141592653589793238462` in one part of my code, but `3.14159285358979323846 2` in another place. Not only is there a bug there (the two values are actually different), but it's an extraordinarily difficult bug to detect.
- Along those same lines, hard-coding these values can make it difficult to detect when you've used the wrong magic number in a given expression, particularly when you have multiple magic numbers in your code.
- Depending on how complicated these values are, they can clutter the code. For example, which of the following two expressions is easier to glance at and understand? `3.14159265358979323846*r**2`, or `pi*r**2`?
- It may not seem like it, but these values actually also encode other assumptions we're making, which may change later. For example, encoded in that magic number is the degree of precision we use to represent it; if I wanted to change that later, I would need to change it in multiple places.

It is also important to keep in mind, when refactoring code to remove magic numbers, that some names are better than others. For example, the following is not any more useful than using `7` as a magic number:

```
seven = 7
```

but the following could be useful:

```
days_per_week = 7
cards_in_hand = 7
max_number_of_players = 7
```

This is also especially relevant for looping variables that aren't simply counters. While it can be tempting to use generic one-character names for any looping variables, it is often possible to clarify things by using more descriptive names. For example, which of the following two pieces of code is easier to read?

```
def longest_word_in_file(fname):
    longest = 0
    with open(fname) as f:
        for i in f:
            for j in i.split():
                if len(j) > longest:
                    longest = len(j)
    return longest
```

```
def longest_word_in_file(fname):
    longest = 0
    with open(fname) as f:
        for line in f:
            for word in line.split():
                if len(word) > longest:
                    longest = len(word)
    return longest
```

2.2.3) One Purpose For Each Variable

In general, within each function or module, distinct logical entities should be given distinct names. It is generally a bad idea to reuse the same name to mean different things within a single function, for example.

As a rule of thumb, don't reuse parameters, and don't reuse variable names. Variables are not a scarce resource in Python, so you can/should introduce them freely, give them good names, and just stop using them when you stop needing them.

In general, someone reading your code will likely be confused if a name that used to mean one thing suddenly starts meaning something different a few lines later (or if the name of a parameter is reused within the body of a function to mean something else).

2.3) Documentation Matters

The kindest thing you can do for future people reading your code (which includes not only your 6.009 staff, but also your future self!) is to document your code, and to do so judiciously.

In Python, there are at least two different types of documentation (docstrings and comments), which we'll discuss in more detail below. While these structures can, in some sense, be used interchangeably, they are often used for different things in practice.

Good documentation should not only make the code easier to understand (by documenting non-obvious features of the code), but also safer against bugs (because important assumptions have also been documented).

2.3.1) Docstrings

One important kind of documentation is a docstring, which is a regular Python string on the first line of a function, and which serves as documentation for that function (classes and modules can also have docstrings, but here we'll focus on functions). While any string is fine, it is conventional to use triple-quotes (`"""`) around docstrings, even for one-line docstrings.

Here is a function with a (relatively) useful docstring:

```
def zeller(year, month, day):
    """
    Compute the day of the week on which a particular date fell.

    Uses Zeller's algorithm to compute the result
        (see https://en.wikipedia.org/wiki/Zeller%27s_congruence)

    Parameters:
     * year (int) : the year of interest; requires year != 0; negative values
         correspond to years BC, and positive values to years AD.

     * month (int) : the month of interest; requires 1<=month<=12; January is
         represented by 1, and December by 12

     * day (int) : the day of interest, taken as a 1-indexed value within
         the given month; must be a day that exists within the given month

    Returns:
      A 1-character string representing the day of the week on which the given
      date fell:
        * 'M' for Monday
        * 'T' for Tuesday
        * 'W' for Wednesday
        * 'R' for Thursday
        * 'F' for Friday
        * 'S' for Saturday
        * 'U' for Sunday
```

```
    """
    # code here ...
```

Docstrings are an important type of documentation, as they are used to document not only the behavior/purpose of the function/module/class, but also assumptions about the inputs and outputs. Docstrings are also used internally by Python; they are stored as an attribute called `__doc__` inside the object, and they are displayed when a user invokes the `help` function on the object. For example, `print(zeller.__doc__)` would print the docstring, and `help(zeller)` would display the docstring.

The example above provides a lot of useful information about that function. By contrast, the following docstring (for the same function) is not quite as useful:

```
def zeller(year, month, date):
    """Return the day of the week."""
    # code here ...
```

While we aren't going to care too much about the particular formatting of these docstrings (there are many different conventions), including them is really important from a stylistic perspective, and the best docstrings will generally contain:

- a brief one-sentence description of the function's behavior
- if necessary, a longer description of the behavior
- a description of inputs (including types as well as any other expectations about those values)
- a description of effects (including return value/type, any exceptions that might be raised, or other side effects of the function.

2.3.2) Comments

Inline comments can also be useful, particularly when they describe aspects of the code that are not obvious. It is often reasonable to assume that the reader can figure out *what* the code does; it is more useful to explain *why* (though in the case of really non-standard structures or design decisions, it is good to leave a comment explaining how that structure accomplishes its task, in addition to your justification for structuring things that way).

As you are writing code, you should document the pieces of the code that are non-obvious, but let the code speak for itself where it can.

Some comments are not useful at all. For example, direct translations of code into English do nothing to improve understanding, and so should be avoided (it is safe to assume your reader knows Python):

```
v = 5  # set v to 5
while n != 1:  # test whether n is 1
    i += 1  # increment i
    mylist.append(n)  # add n to mylist
```

By contrast, the following comments all provide useful information that does not otherwise exist in the code:

```
velocity = 5  # meters / second

sum = n*(n+1)/2  # Gauss's formula for the sum of 1...n

# here we're using the approximation that sin(x) ~= x when x is small
moon_diameter_meters = moon_distance_meters * apparent_angle_radians
```

And the comments in the following function describe high-level organization that may not be apparent from the code itself:

```
def flatten_list(input_list):
    """
    Create a new "flat" list from a nested list.

    For example:
        >>> flatten_list([1, 2, [3, [4, [[[5]]]], 6, [7, 8]], 9])
        [1, 2, 3, 4, 5, 6, 7, 8, 9]

    Parameters:
     * input_list (list): a list of elements, possibly containing
         arbitrarily-nested lists

    Returns:
      A list containing all of the non-list elements from the given input_list,
      but with any nesting of lists removed.
    """
    # make a (shallow) copy to avoid accidentally mutating input_list
    flat_list = input_list[:]
    while True:
        for ix, element in enumerate(flat_list):
            if isinstance(element, list):
                # break out of the for loop, to update flat_list
                break
        else:
            # if we're here, we did not break out of the for loop (i.e., we did
            # not find a list inside of flat_list), so we are done!
            return flat_list

        # because of the for/else structure, if we reach this point, we found a
        # nested list (ix contains the index of the nested list within
        # flat_list, and element contains the list itself).  So we'll replace
        # that list with its contents.
        flat_list[ix:ix+1] = element
```

Good variable names can reduce the need for certain kinds of comments. For example, choosing the second of the following variable names is preferred, and makes the comment from the first version unnecessary:

```
n = 60*60*24  # number of seconds in a day
```

```
seconds_per_day = 60*60*24
```

2.3.3) Commented-out Code

During the development process, it is common to "comment out" various sections of code while working, in order to try new things or to help with debugging.

This is a really great thing to do, and you should definitely do it! But when your code is ready (in our context, this means ready to be submitted to the web site; in other contexts, it might mean that it's ready to be released to the world), you should remove sections of unused code that have been commented out.

2.4) Generality Wins

It is best to define functions and programs generally where possible, and to handle specific cases by providing the proper inputs to the function.

For example, the `square` function is not defined in the `math` module, nor is `cube`. This is because they are both specific cases for the `pow` function (exponentiation), which *is* in the `math` module.

Writing a single general-purpose function results in code that is easier to read, write, and debug, in part because it helps to avoid the duplicate code that often results from creating several specific (but related) functions.

2.4.1) Avoid Superfluous Special Cases

Programmers are often tempted to write special code to deal with what seem like special cases (parameters that are 0, for example, or empty lists, or empty strings). However, it is often the case that an existing general procedure, or a slight variation thereof, could handle the task just as well.

For example, consider the following piece of code, designed to return the length of the longest string in its input:

```
def longest_string_length(words):
    if len(words) == 0:
        return 0

    longest = 0
    for word in words:
        if len(word) > longest:
            longest = len(word)
    return longest
```

If we look closely at this function, we find that the first `if` statement is unnecessary. The function would behave the same way, with or without it: if the words list is empty, then the `for` loop simply does nothing, and `0` is returned anyway. As such, it would be better to omit that first `if` statement altogether.

This issue is also especially relevant in recursive functions, where it can be tempting to define more bases cases than are actually necessary. For example, consider the following function, designed to compute the sum of all elements in a list recursively:

```
def sum_all(values):
    if len(values) == 0:
        return 0
    elif len(values) == 1:
        return values[0]
    else:
        return values[0] + sum_all(values[1:])
```

It turns out that the recursive case here would handle the case where `values` has exactly one element in it just fine! So that middle case is unnecessary, and we could instead have only included the first base case (handling the case of an empty `values` list).

If you find yourself writing an if statement for a special case, stop what you're doing, and think about the general-case code, either to confirm that it can actually already handle the special case you're worrying about (which is often true!), or try to think of a way to make your general case handle that special case as well.

Writing broader, general-case code pays off. It results in shorter functions, which is easier to understand and has fewer places for bugs to hide, and it is also likely to be safer from bugs because it makes fewer assumptions about the values it is working with.

2.4.2) Special Cases for the Sake of Efficiency

Programmers often justify handling special cases separately with a belief that it increases the overall performance of the method, by returning a hardcoded answer for a special case right away. For example, when writing a sort algorithm, it can be tempting to check whether the size of the list is 0 or 1 at the very start of the method, since you can then return immediately with no need to sort at all. Or if the size is 2, just do a comparison and a possible swap.

These optimizations might indeed make sense, but it is a good idea only to implement such things when you have evidence that they would actually make an appreciable difference in the speed of the program.

Continuing the example from above, if our sorting method is almost never called with these special cases, then adding code for them just adds complexity, overhead, and hiding-places for bugs, without any practical improvement in performance. As a general rule, you should try to write a clean, simple, general-case algorithm first, and optimize it later, only if it would actually help.

To quote Donald Knuth:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

2.5) Plan for Change

Oftentimes, when writing programs, you may find that the requirements for the program you're writing may change (or you may find that the problem you *actually* wanted to solve isn't the one solved by the program you're writing). As such, whenever possible, it is important to attempt to make programs that are as easy to change as possible, should the need arise.

The following are some general ideas that you can keep in mind as you work, in order to make sure that your code is amenable to change.

2.5.1) Avoid Monolithic Functions

It can be tempting, when writing a complicated program, to implement the entire program in a single function. However, it is a good idea to try to separate logically-independent suboperations from each other (usually, into smaller helper functions), for a few reasons:

- Doing so usually makes the larger program easier to reason about, since, if these helper functions have good names, we are able to describe the large program in terms of higher-level operations.
- Doing so also means that if we find another use for one of the suboperations, we can make use of that helper function elsewhere in our program without needing to duplicate that code.
- Smaller programs are generally easier to debug than larger programs, so testing and debugging several small functions is generally more straightforward than testing and debugging a single more complicated function.

2.5.2) Store Local Information Locally

In general, it is a good idea, when information is only needed in one small portion of a program, to make sure that that information is only available in that one portion of the code. Variables that are local to a function are one example of making good use of this principle.

This also suggests that it is a good idea to avoid using global variables altogether. This page has some good examples of the dangers of global variables.

Global variables are often used so that functions can have access to the same piece of data across multiple calls; but there are usually better ways of doing that, that manage to avoid some or all of the dangers associated with global variables. For example, if you want a single function to have access to the same piece of data across multiple calls to that function, you could consider using a closure. If you want multiple functions to have access to the same piece of data, you could consider creating a class.

2.5.3) Return Results, Don't Print Them

Consider the following function:

```
def count_long_words(word_list):
    num_long_words = 0
    for word in word_list:
        if len(word) > LONG_WORD_LENGTH:
            num_long_words += 1
    print(num_long_words)
```

This function is not particularly amenable to change (and, consequently, programs making use of it are not particularly amenable to change), mostly because it sends its output to the terminal via `print`. That means that if you want to use it in another context (where the number is needed for some other purpose, like computation rather than human eyes) it would have to be rearranged.

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results. The sole exception here is debugging output, which can of course be printed to the console. But that kind of output shouldn't be a part of your design, only a part of how you debug your design.