Composition

Overview

The ease by which other *people* can read and understand a program (often called "readability" in software engineering) is perhaps the most important quality of a program. Readable programs are used and extended by others, sometimes for decades. For this reason, we often repeat in CS 61A that programs are written to be read by humans, and only incidentally to be interpreted by computers.

In CS 61A, each project has a composition score that is graded on the *style* of your code. This document provides some guidelines. A program is composed well if it is concise, well-named, understandable, and easy to follow.

Excellent composition does not mean adhering strictly to prescribed style conventions. There are many ways to program well, just as there are many styles of effective communication. However, the following guiding principles universally lead to better composition of programs:

- **Names**. To a computer, names are arbitrary symbols: "xegyawebpi" and "foo" are just as meaningful as "tally" and "denominator". To humans, comprehensible names aid immensely in comprehending programs. Choose names for your functions and values that indicate their use, purpose, and meaning. See the lecture notes section on choosing names (http://www.composingprograms.com/pages/13-defining-new-functions.html#choosing-names) for more suggestions.
- **Functions**. Functions are our primary mechanism for abstraction, and so each function should ideally have a single job that can be used throughout a program. When given the choice between calling a function or copying and pasting its body, strive to call the function and maintain abstraction in your program. See the lecture notes section on composing functions (http://www.composingprograms.com/pages/14-designing-functions.html#designing-functions) for more suggestions.
- **Purpose**. Each line of code in a program should have a purpose. Statements should be removed if they no longer have any effect (perhaps because they were useful for a previous version of the program, but are no longer needed). Large blocks of unused code, even when turned into comments, are confusing to readers. Feel free to keep your old implementations in a separate file for your own use, but don't turn them in as your finished product.
- **Brevity**. An idea expressed in four lines of code is often clearer than the same idea expressed in forty. You do not need to try to minimize the length of your program, but look for opportunities to reduce the size of your program substantially by reusing functions you have already defined.

Names and variables

Variable and function names should be *self-descriptive*:

Good

```
goal, score, opp_score = 100, 0, 0
greeting = 'hello world'
is_even = lambda x: x % 2
```

Bad

```
a, b, m = 100, 0, 0
thing = 'hello world'
stuff = lambda x: x % 2
```

Indices and mathematical symbols

Using one-letter names and abbreviations is okay for indices, mathematical symbols, or if it is obvious what the variables are referring to.

Good

```
i = 0         # a counter for a loop
x, y = 0, 0   # x and y coordinates
p, q = 5, 17  # mathematical names in the context of the question
```

In general, `i` , `j` , and `k` are the most common indices used.

'o' and 'l'

Do not use the letters 'o' and 'l' by themselves as names:

Bad

```
o = O + 4     # letter 'O' or number 0?
l = l + 5     # letter 'l' or number 1?
```

Unnecessary variables

Don't create unnecessary variables. For example,

Good

```
return answer(argument)
```

Bad

```
result = answer(argument)
return result
```

However, if it is unclear what your code is referring to, or if the expression is too long, you *should* create a variable:

Good

```
divisible_49 = lambda x: x % 49 == 0
score = (total + 1) // 7
do_something(divisible_49, score)
```

Bad

```
do_something(lambda x: x % 49 == 0, (total + 1) // 7)
```

Profanity

Don't leave profanity in your code. Even if you're really frustrated.

Bad

```
eff_this_class = 666
```

Naming convention

Use `lower_case_and_underscores` for variables and functions:

Good

```
total_score = 0
final_score = 1

def mean_strategy(score, opp):
    ...
```

Bad

```
TotalScore = 0
finalScore = 1

def Mean_Strategy(score, opp):
    ...
```

On the other hand, use `CamelCase` for classes:

Good

```
class ExampleClass:
    ...
```

Bad

```
class example_class:
    ...
```

Spacing and Indentation

Whitespace style might seem superfluous, but using whitespace in certain places (and omitting it in others) will often make it easier to read code. In addition, since Python code depends on whitespace (e.g. indentation), it requires some extra attention.

Spaces vs. tabs

Use spaces, not tabs for indentation. Our starter code always uses 4 spaces instead of tabs. If you use both spaces and tabs, Python will raise an `IndentationError`.

Many text editors, including Sublime and Atom, offer a setting to automatically use spaces instead of tabs.

Indent size

Use 4 spaces to denote an indent. Technically, Python allows you to use any number of spaces as long as you are consistent across an indentation level. The conventional style is to use 4 spaces.

Line Length

Keep lines under 80 characters long. Other conventions use 70 or 72 characters, but 80 is usually the upper limit. 80 characters is not a hard limit, but exercise good judgement! Long lines might be a sign that the logic is too much to fit on one line!

Double-spacing

Don't double-space code. That is, do *not* insert a blank line in between lines of code. It increases the amount of scrolling needed and goes against the style of the rest of the code we provide.

One exception to this rule is that there should be space between two functions or classes.

Spaces with operators

Use spaces between `+` and `-`. Depending on how illegible expressions get, you can use your own judgement for `*`, `/`, and `**` (as long as it's easy to read at a glance, it's fine).

Good

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

Bad

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

Spacing lists

When using tuples, lists, or function operands, leave one space after each comma `,`:

Good

```
tup = (x, x/2, x/3, x/4)
```

Bad

```
tup = (x,x/2,x/3,x/4)
```

Line wrapping

If a line gets too long, use parentheses to continue onto the next line:

Good

```
def func(a, b, c, d, e, f,
         g, h, i):
    # body

tup = (1, 2, 3, 4, 5,
       6, 7, 8)
names = ('alice',
         'bob',
         'eve')
```

Notice that the subsequent lines line up with the *start* of the sequence. It can also be good practice to add an indent to imply expression continuation; use whichever format expresses the line continuation most clearly.

> Good

```
total = (this_is(a, very, lengthy) + line + of_code
            + so_it - should(be, separated)
            + onto(multiple, lines))
```

Blank lines

Leave a blank line between the end of a function or class and the next line:

> Good

```
def example():
    return 'stuff'

x = example() # notice the space above
```

Trailing whitespace

Don't leave whitespace at the end of a line.

Control Structures

Boolean comparisons

Don't compare a boolean variable to `True` or `False`:

> Bad

```
if pred == True:   # bad!
    ...
if pred == False:  # bad!
    ...
```

Instead, do this:

Good

```
if pred:           # good!
    ...
if not pred:       # good!
    ...
```

Use the "implicit" `False` value when possible. Examples include empty containers like `[]`, `()`, `{}`, `set()`.

Good

```
if lst:       # if lst is not empty
    ...
if not tup:   # if tup is empty
    ...
```

Checking `None`

Use `is` and `is not` for `None`, not `==` and `!=`.

Redundant if/else

Don't do this:

Bad

```
if pred:            # bad!
    return True
else:
    return False
```

Instead, do this:

Good

```
return pred         # good!
```

Likewise:

Bad

```
if num != 49:
    total += example(4, 5, True)
else:
    total += example(4, 5, False)
```

In the example above, the only thing that changes between the conditionals is the boolean at the end. Instead, do this:

Good

```
total += example(4, 5, num != 49)
```

In addition, don't include the same code in both the `if` and the `else` clause of a conditional:

Bad

```
if pred:            # bad!
    print('stuff')
    x += 1
    return x
else:
    x += 1
    return x
```

Instead, pull the line(s) out of the conditional:

Good

```
if pred:            # good!
    print('stuff')
x += 1
return x
```

while vs. if

Don't use a `while` loop when you should use an `if`:

Bad

```
while pred:
    x += 1
    return x
```

Instead, use an `if`:

> Good

```
if pred:
    x += 1
    return x
```

Parentheses

Don't use parentheses with conditional statements:

> Bad

```
if (x == 4):
    ...
elif (x == 5):
    ...
while (x < 10):
    ...
```

Parentheses are not necessary in Python conditionals (they are in other languages though).

Comments

Recall that Python comments begin with the `#` sign. Keep in mind that the triple-quotes are technically strings, not comments. Comments can be helpful for explaining ambiguous code, but there are some guidelines for when to use them.

Docstrings

Put docstrings only at the top of functions. Docstrings are denoted by triple-quotes at the beginning of a function or class:

> Good

```
def average(fn, samples):
    """Calls a 0-argument function SAMPLES times, and takes
    the average of the outcome.
    """
```

You should not put docstrings in the middle of the function -- only put them at the beginning.

Remove commented-out code

Remove commented-out code from final version. You can comment lines out when you are debugging but make sure your final submission is free of commented-out code. This makes it easier for readers to identify relevant portions of code.

Unnecessary comments

Don't write unnecessary comments. For example, the following is bad:

Bad

```
def example(y):
    x += 1             # increments x by 1
    return square(x)   # returns the square of x
```

Your actual code should be *self-documenting* -- try to make it as obvious as possible what you are doing without resorting to comments. Only use comments if something is not obvious or needs to be explicitly emphasized.

Repetition

In general, **don't repeat yourself** (DRY). It wastes space and can be computationally inefficient. It can also make the code less readable.

Do not repeat complex expressions:

Bad

```
if a + b - 3 * h / 2 % 47 == 4:
    total += a + b - 3 * h / 2 % 47
    return total
```

Instead, store the expression in a variable:

Good

```
turn_score = a + b - 3 * h / 2 % 47
if turn_score == 4:
    total += turn_score
    return total
```

Don't repeat computationally-heavy function calls either:

Bad

```
if takes_one_minute_to_run(x) != ():
    first = takes_one_minute_to_run(x)[0]
    second = takes_one_minute_to_run(x)[1]
    third = takes_one_minute_to_run(x)[2]
```

Instead, store the expression in a variable:

Good

```
result = takes_one_minute_to_run(x)
if result != ():
    first = result[0]
    second = result[1]
    third = result[2]
```

Semicolons

Do not use semicolons. Python statements don't need to end with semicolons.

Generator expressions

Generator expressions are okay for simple expressions. This includes list comprehensions, dictionary comprehensions, set comprehensions, etc. Generator expressions are neat ways to concisely create lists. Simple ones are fine:

Good

```
ex = [x*x for x in range(10)]
L = [pair[0] + pair[1]
     for pair in pairs
     if len(pair) == 2]
```

However, complex generator expressions are very hard to read, even illegible. As such, do not use generator expressions for complex expressions.

Bad

```
L = [x + y + z for x in nums if x > 10 for y in nums2 for z in nums3 if y > z]
```

Use your best judgement.

CS 61A (/~cs61a/sp19/)

Weekly Schedule (/~cs61a/sp19/weekly.html)

Office Hours (/~cs61a/sp19/office-hours.html)

Staff (/~cs61a/sp19/staff.html)

Resources (/~cs61a/sp19/resources.html)

Studying Guide (/~cs61a/sp19/articles/studying.html)

Debugging Guide (/~cs61a/sp19/articles/debugging.html)

Composition Guide (/~cs61a/sp19/articles/composition.html)

Policies (/~cs61a/sp19/articles/about.html)

Assignments (/~cs61a/sp19/articles/about.html#assignments)

Exams (/~cs61a/sp19/articles/about.html#exams)

Grading (/~cs61a/sp19/articles/about.html#grading)