IFB104 Code Presentation Guide

Overview

Writing computer program, scripting or mark-up language code is no different from any other style of writing. Its purpose is to communicate concisely and clearly. Any computer language document must communicate effectively in two ways:

- Telling the computer what to do. The "executable" part of the program, script or mark-up tells the computer what actions you want performed. We know if this has been done properly if the document gets the computer to do or display what we intended.
- Informing other programmers. These may be your immediate colleagues if you're working in a team or someone else who reads your code many months or years from now, perhaps with the intention of modifying or extending your code's behaviour.

Our concern here is with the second of these. Part of the profession of "building IT systems" is to create a well-presented computer language document. Over the years, programmers have developed style guides and conventions for most pro- gramming, scripting and mark-up languages, and you will be able to find many of these online. For the purposes of IFB104, however, we do not want to be too pre- scriptive. Our main goal is that you follow a consistent style that meets the criteria listed below. The examples below are for Python program code, but the same prin- ciples should be applied to any other code you write, including SQLite scripts and HTML/CSS documents.

General Principles

As a general guide, consider that whether you're writing a text message, a novel, a technical report or a computer program, there is one rule that *always* applies:

> **Whatever is quick and convenient for the author is slow and awkward for the reader.**

For instance, frequent use of abbreviations allows you to type quickly, but makes your writing harder to understand. Studies have shown that even expert "texters" are slower to read and comprehend heavily-abbreviated SMS messages than the equivalent English text.

Another important rule, highly relevant to long-lived computer programs that are often modified during their lifetime, is:

> **Code is written once but read many times.**

This means that any extra effort expended during writing will be paid back to us lat- er on.

While these principles are mainly relevant to large-scale software development (ra- ther than small university assignments) our goal is to get you into the habit of think- ing about your code's presentation at an early stage.

Code Presentation Criteria

You should think of your code's presentation in terms of the following criteria.

1. Clear, uncluttered layout

Your code should be laid out neatly and consistently. To a large extent Python pro- grammers are forced to adopt a particular layout because indentation is significant in the language. Nevertheless you should give thought to choices of vertical white space, line breaks and commenting. Many people frown upon overly wide lines, which inhibit readability and make it necessary to scroll back and forth horizontally. In general lines should not be wider than about 80 characters. (You can see the cur- rent column number in the bottom of IDLE's editing window.)

Consideration should also be given to consistent horizontal spacing, in order to avoid cluttered code and improve its readability. Always put a space after a comma, and surround operators with spaces, e.g., "`7 * 9 != 9 + 7`", rather than "`7*9!=9+7`". However, don't include a space between a function/method name and the opening parenthesis.

Lines of related code should be blocked together to improve readability. Blocks can be formed simply by adding appropriate line breaks.

2. Meaningful choices of variable and function identifiers

You usually have a free choice of variable and function names in your code. You should make good use of this to explain the *purpose* of the variable or function, i.e., its role in the computation. Consider the following code segment.

```
def a(b):
    x = len(b) // 2
    y = b[:x]
    z = b[x:]
    return [y, z]
```

This code is hard to understand because none of the identifiers explain the func- tion's or the variables' purposes.

Now consider exactly the same code with well-chosen identifiers.

```
def split_in_half(text):
    midpoint = len(text) // 2
    first_half = text[:midpoint]
    second_half = text[midpoint:]
    return [first_half, second_half]
```

Even without comments this code is now easy to understand. We can see from the function signature that its purpose is to split some text in half. The role of each vari- able is made clear by its name, e.g., the first one finds the midpoint of the given text, and so on. Choosing sensible variable and function names has a *dramatic* impact on the understandability of your code.

Avoid the use of abbreviations when it comes to variables and function names.

Function/method names should be verbs, or 'doing' words which reflect the pur- pose of the function/method. Variable names should reflect the value/s intended to be stored in them.

IMPORTANT NOTE: Some old-fashioned programming languages, and even text- books, encourage the use of single-character identifiers, especially for loop varia- bles, typically '`i`', '`j`', '`k`' and so on. These languages usually do not offer Python's ability to iterate over all values in a sequence (e.g., a list or string). Thus, where in Python we would write

```
for letter in word:
    print(letter)
```

to print each letter in a given string, each on one line, older languages force the pro- grammer to introduce an "index" variable and express this computation in a form equivalent to:

```
for i in range(len(word)):
    print(word[i])
```

Notice how much more obscure the second version is. The index variable is unnec- essary and the meaningless choice of identifier '`i`' is unhelpful. In Python each vari- able introduced has a specific purpose and should be given a name describing the values it contains. **Code like the second `for` loop above, containing unneces- sary, single- character loop variable names, should be avoided.**

3. Concise, helpful commenting

Your program code should be commented to explain features that are **not obvious from the code itself**. In particular, you can assume that the reader can understand *what* the code does by inspecting it, but will not necessarily understand *why*. Your comments should thus explain the **purpose** of obscure code segments. However, you should not clutter the code with comments that add no helpful information for the reader.

For instance, the following comment is useless and should **not** be included, because it does not tell us anything we can't see from the code itself.

```
amount = amount + 1 # add one to amount
```

On the other hand, the comment below (taken from one of the IFB104 lecture demonstrations) makes the purpose of the following statements clearer than just the code itself.

```
# Draw the middle circle
penup()
home()
pendown()
color("orange")
dot(middle_radius * 2)
```

The comment in this case tells us *what* the code does (but *how* the code does this job can be seen from the code itself and needs little or no commenting).

A comment should precede (i.e., come before) the relevant code block and be indented to the same level as the code to which it refers.

4. No magic numbers

A "magic number" is a constant in your code whose purpose and choice of value is not immediately apparent. Although the term is normally applied to numerical val- ues, it is equally applicable to any literal value, regardless of its type. Magic num- bers make code harder to understand. They also make code harder to maintain be- cause if the number appears in several places, and we want to change it, then we have to search the code for all of its occurrences.

We can eliminate magic numbers in either of two ways:

- By explaining the number's purpose in a comment where it appears.
- By giving the number a meaningful name.

The first of these is acceptable if the number only appears in a few places in the code. However, if the number is used in several places it is better to name it, be- cause this means only one code change is needed to modify the number.

For example, the following code segment is obscure due to the use of a poor variable name, no commenting and the unexplained magic number '7'.

```
x = (x + 1) % 7
```

The following version is longer but much clearer.

```
days_in_week = 7 # days are numbered from 0 to 6

…

day = (day + 1) % days_in_week # advance to next day
```

(Observation: Should the '1' in the 'day' example above or the '2' in the 'circle' example previously shown be considered as magic numbers? Arguably not because their purpose is very obvious in both cases. The number 1 is clearly being used to

increment variable `day` and the number 2 is clearly being used to double a radius to produce a diameter. It would clutter the code to give special names to these 'obvi- ous' uses. However, if in doubt, it never hurts to add a *brief* comment. Also, we ac- cept that it's very difficult to write Turtle graphics code without using a lot of specif- ic numbers for coordinates and distances, so these "magic" numbers are usually ac- ceptable since these numbers are used once only and for a clear purpose, e.g., to go to a specific location on screen.)

5. No unnecessary duplication of code segments

You should aim to develop as concise a solution as possible. For instance, we learn in this teaching unit how functions allow us to reuse code segments that would oth- erwise appear more than once in your program. You should aim to use them to eliminate duplicated code wherever possible. Similarly, loops should be used rather then repeating similar code many times.

6. Correct grammar and spelling

You should present your computer language code as professionally as you would any other document that will be seen by your colleagues. Code that contains spelling and grammatical errors in either the variable/function names or the com- ments gives a poor impression.