



Queensland University of Technology
Brisbane Australia

IFB104 Code Presentation Guide

Overview

Writing computer program or scripting language code is no different from any other style of writing. Its purpose is to communicate concisely and clearly. A computer language program or script must communicate effectively in two ways:

- Telling the computer what to do. The “executable” part of the program or script tells the computer what actions you want performed. We know if this has been done properly if the program “works”.
- Informing other programmers. These may be your immediate colleagues if you’re working in a team or someone else who reads your program 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 program or script. Over the years, programmers have developed style guides and conventions for most programming languages, and you will be able to find many of these for Python online. For the purposes of IFB104, however, we do not want to be too prescriptive. Our main goal is that you follow a consistent style that meets the criteria listed below.

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 later on.



While these principles are mainly relevant to large-scale software development (rather than small IFB104 assignments) our goal is to get you into the habit of thinking 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 programmers are forced to adopt a particular layout because indentation is significant in the language. Nevertheless you should give thought to choices of line breaks and commenting. Many people frown upon overly wide lines, which inhibit readability and make it necessary to scroll back and forth. In general lines should not be wider than about 80 characters. (You can see the current column number in the bottom of IDLE's editing window.)

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 calculation. Consider the following code segment.

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

This code is very hard to understand because none of the identifiers explain the function'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 variable 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.

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

```
for letter in word:
    print letter
```

to print each letter in a given string on one line, older languages force the programmer 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 unnecessary and the meaningless choice of identifier is unhelpful. In Python each variable introduced has a specific purpose and should be given a name describing the values it contains. **Code like the second for loop above, containing unnecessary, 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 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).

4. No magic numbers

A “magic number” is a number in your code whose purpose and choice of value is not immediately apparent. Although the term is normally applied to numerical values, it is equally applicable to any literal value (constant), regardless of its type. Magic numbers make code harder to understand. They also make it harder to maintain because 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 one place in the code. However, if the number is used in several places it is better to name it, because 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 ‘obvious’ uses. However, if in doubt, it never hurts to add a *brief* comment. Also, we accept that it’s often difficult to write Turtle graphics code without using a lot of specific numbers for coordinates and distances, but this is usually acceptable since these numbers are used once and for a clear purpose, e.g., to go to a specific location on screen.)



Queensland University of Technology
Brisbane Australia

5. No unnecessary duplication of code segments

You should aim to develop as concise a solution as possible. For instance, we have learnt in this unit how functions allow us to reuse code segments that would otherwise appear more than once in your program, so you should aim to use them to eliminate duplicated code wherever possible. Similarly, loops should be used rather than repeating code many times.

6. Correct grammar and spelling

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