Lecture 3.2 – Writing High-Quality Code

M30299 Programming

School of Computing
University of Portsmouth

Introduction to lecture

- This lecture will consider how to write high-quality code.
- We take “high-quality code” to mean the following:
    - code that is **readable**; and
    - code that is **correct**.
- We will consider each of these in turn, using example exercises similar to those in the practical worksheets as motivation.

What is readable code?

- What do we mean by readable code, and why is it significant?
- We will consider program code to be readable if it can be easily understood by anyone who is:
  - familiar with programming in the language (here, Python); but
  - not necessarily familiar with what the code is meant to do.
- Writing code that is readable is important since:
  - Software in industry is often written by **teams** of people.
  - Successful software is not just written; it is **maintained** (modified and extended), over many years, by many people.

How to write readable code

- It is not so difficult to write readable code. (And there is no reason at all to first write unreadable code and then make it readable!)
- Restricting ourselves to the programming concepts seen so far, the following are important:
    - good use of variables and variable (and function) **names**;
    - good use of **whitespace**;
    - good **documentation**; and
    - avoiding overly complicated (and/or repetitive) code.
- Let’s consider some of these ...

Which function is more readable?

```
def cost():
    a=float(input("Enter width: "))
    b=float(input("Enter length: "))
    c=float(input("Enter height: "))
    print ("Cost of cake is",(a* b * c)*3.5 +60 ,"pence")

def cost_of_cake():
    width = float(input("Enter width: "))
    length = float(input("Enter length: "))
    height = float(input("Enter height: "))
    volume = width * length * height
    cake_mix_cost_per_cm3 = 3.5
    cherry_cost = 6
    number_of_cherries = 10
    total_cost = 3.5 * volume + cherry_cost * number_of_cherries
    print("Cost of cake is", total_cost, "pence")
```

Choosing good names

- The name of a variable or function has first to be **legal**:
  - it must begin with a letter or underscore, and only consist of letters, digits and underscores.
  - (so `xyz_137123` is legal but `1p` and `num 1` are not);
  - it must not be a **keyword** such as `def` or `for`;
  - it's best to avoid the names of built-in functions such as `sum`.
- Use `snake_case` (lowercase with underscores) which is a Python standard.
- When writing code, take time to **think about names**:
  - use informative names (to help explain what the code means);
  - try to avoid abbreviations like `diam`;
  - single letter names are ok in a few places (`x` & `y` for coordinates).

Using whitespace (tabs, spaces, blank lines)

- You have already understood that the “bodies” of functions and loops **must** be **indented** consistently for them to work.
- Stick to standard conventions for other uses of whitespace:
  - Leave two blank lines between function definitions.
  - Use a single space either side of an assignment symbol and operator, but not before or after containing brackets; e.g.:

    ```
    area = math.pi * (diameter / 2) ** 2
    ```

  - Use a single space after commas; e.g.:

    ```
    print("There are", pizzas_left, "pizzas left.")
    ```

- Do **not** put whitespace between function names and brackets, or before colons, as in: `for i in range (5) :`

Avoiding long lines of code

- Use 79 characters as a limit for each line of code (another Python standard).
- This will make the program text easier to read, and also mean that code will not be cropped or wrapped when you print it.
- A long statement can be split across two lines using `()` or `\`. E.g.,

```
pizza_cost = area * (dough_cost + cheese_cost
                     + ham_cost + tomato_cost)
sandwich_cost = (2 * bread_cost) + butter_cost \
                + bacon_cost + lettuce_cost + tomato_cost
```

- Notice that the second line of each statement can be indented to any amount, and here we indent to maximise readability.

Documentation (comments)

- Another technique you can use to produce readable code is to **document** it, using English text.
- Documentation of code can take the form of **comments**, which appear to the right of the # symbol.
- We have used comments so far to identify the contents, author and date of our Python files.
- Comments are also useful to explain to the reader something that **might not be obvious** on reading just the code; e.g.,

```
# apply Pythagoras's theorem
distance = ((x_2 – x_1) ** 2 + (y_2 – y_1) ** 2) ** 0.5
```

Documentation (comments)

- Many people think that: more comments = better code.
- This is wrong—if your code is well written, it should be understandable without the need for too many comments.
- Some comments are pointless, and too many comments can make the code **more difficult** to read. For example:

```
def cost_of_cake():
    width = float(input("Enter width: "))  # get width from user
    length = float(input("Enter length: "))  # get length from user
    height= float(input("Enter height: "))  # get height from user
    volume = width * length * height  # calculate volume
    . . .
```

Correct code: testing

- Consider the following question on worksheet 1:

  > Write a function called `dollars_to_pounds` which converts an amount in dollars entered by the user to a corresponding amount in pounds. Assume that the exchange rate is 1.35 dollars to the pound.

- A common incorrect solution gives the following behaviour:

```
Enter an amount in dollars: 1.35
The corresponding amount in pounds is 1.8225000000000002
Enter an amount in dollars: 1
The corresponding amount in pounds is 1.35
```

Testing your code

- These errors were due to not fully understanding the task.
- The critical part to understand was that:

  1.35 dollars = 1 pound

- This fact leads to two things:
  - the first piece of **test data** to use could be 1.35 dollars, and you would expect 1.0 pounds as the output;
  - (after some thought) the correct assignment statement to use in the code is `pounds = dollars / 1.35`.
- Notice that you can work out appropriate test data **before** you've written the function (i.e. as you are understanding the task).
- And, if your function doesn't give the expected output, it's wrong!

Testing and test data

- A good approach for attempting a (short) programming problem:
    1. make sure that you understand the task, in particularly by:
    2. thinking of (and writing down) some appropriate test data and corresponding expected outputs.
    3. develop your solution as a Python function.
    4. test your function on your test data.
    5. repeat steps 3–4 if the function doesn’t give the expected output.