COMP10001

WORKSHOP #6

Shevon Mendis
shevonm@unimelb.edu.au

Documentation

- Simply put, adding documentation to our code is a means of communicating the logic behind our approach when writing code to solve a particular problem

- Why should we document our code?

    - So that others looking at our code are able to understand its purpose and why we've made certain decisions in our approach
        - This is particularly useful for those maintaining the code, as they could use the comments as guide in locating the source of bug (if a bug is found to exist, that is)

    - Helps the author of the code to remember what their train of thought was when they wrote the program, especially if they haven't looked at it for some time

- **Note:** All comments are discarded before a program is run so there is no performance cost incurred by including them

Documentation

- We document our code by using:
  - Comments
    - Single line comments are prefixed using a # symbol
    - For multi-line comments, the comment must be surrounded by a pair of `'''` or `"""`
  - Docstrings for functions
  - Meaningful names for our functions and variables

Commenting Guidelines

- When commenting blocks of code, the comments must appear **before** the aforementioned block of code

☠️ BAD ☠️

```
unique_nums = list(set(numlist))
# remove duplicates from numlist
```

🔥GOOD🔥

```
# remove duplicates from numlist
unique_nums = list(set(numlist))
```

Commenting Guidelines

- Comments should be **concise**

☠️ BAD ☠️

```
# ooo ye formula time 🍆🇦🇺🍆
is_valid = a ** 2 + b ** 2 == c ** 2
```

🔥GOOD🔥

```
# determine if the formula holds
is_valid = a ** 2 + b ** 2 == c ** 2:
```

- **Don't** write comments for code that adequately explains itself

☠️ BAD ☠️

```
# add num1 and num2 and store it in sum
sum = num1 + num2
```

Commenting Guidelines

- If the code block within a loop is performing a substantial amount of actions, then it might be useful to use comments to briefly explain what you're doing at each **distinct** logical step along the way

```
VOWELS = ('A', 'E', 'I', 'O', 'U')

# -- determine the vowel proportion for each word in the list
vowel_counts = {}
for word in word_list:

    # count the number of vowels in each word
    vowel_count = 0
    for letter in word:
        if letter in VOWELS:
            vowel_count += 1

    # calculate the vowel proportion
    vowel_proportion = vowel_count / len(word)

    vowel_counts[word] = vowel_proportion
```

Commenting Guidelines

- Avoid using *inline* comments (comments on the same line as the code) **unless** you're describing the purpose of a variable or constant whose use might not be clear right away

☠️ BAD ☠️

```
player_id = (player_id + 1) % no_of_players # advance to next turn
```

🍕 OKAY 🍕

```
START = (0, 0) # represents the default starting point for the player
```

Naming Conventions

- snake_case vs. camelCase vs. PascalCase
    - Python programmers conventionally using the snake_case format when naming functions and variables. Eg. `my_list` instead of `myList`
    - Aside:
        - camelCase is used only to conform to pre-existing conventions
        - PascalCase is used when defining classes
- When naming functions, methods or variables:
    - They must start with a letter or an underscore
    - They should be in lowercase
    - They can have numbers
    - They can be any length (however, try to keep them short)
    - They can't be the same as a Python keyword

Naming Conventions

- As a guide:
  - When naming variables, think of nouns and ensure that they are **meaningful**
    - Eg. `first_name`, `age`, `address`
  - When naming functions, think of verbs (since functions carry out actions)
    Eg.
    - **get_address(employee_id)** instead of `address_of(employee_id)`
    - `are_related(person1, person2)` instead of `relatives(person1, person2)`

- Single character names:
  - If you're using single characters for variable names, avoid using `l` (lowercase *el*), `I` (uppercase *eye*) and `O` (uppercase *oh*). In certain fonts, the aforementioned characters may be confused with `1`s and `0`s respectively.
  - The only places where it's acceptable to use single characters is for loop variables that work with the `range()` function (Eg. `for i in range(10):`) or if the variable refers to a variable in a mathematical equation (eg. *"write a function that checks if* $a^2 + b^2 = c^2$*..."*)

Magic Numbers

- Magic numbers:
  - Constants which are written into code as literals
  - Why are they bad?
    - As they're coded in as literals, it's hard to understand their meaning
      - Eg.
        ```
        if mark >= 80:
        ```
        What does `80` represent above statement?
    - If the same magic number is used in multiple places within the program, it will make maintenance quite difficult
      - If you need to change the value, you will have to change it in all those places. Forgetting to change it in even one place could result in your program producing incorrect results
      - As a result, they make your code more error-prone

Magic Numbers

- Instead of using magic numbers, store them as global constants at the top of your program and then refer to that variable where necessary in your code

- As a convention, constants must be in uppercase. Eg. `PLANCKS_CONSTANT = 6.626e-34`
  - **Note:** Using global variables for any other context (ie. rather than for storing constants) is considered bad practice

Docstrings

- A Docstring is a comment that describes the purpose of a function and also provides information on how to use it. It can be accessed by calling the `help()` function with your function's name as the argument

- Needed to improve the readability of your code as it provides other developers with the information they need to interact with your function, without having to look at the source code

- Should include:
  - A short description explaining the purpose of the function
  - The input arguments, their types and what they represent
  - What will be returned (if any) and its type

Docstrings

- Blueprint:

```
def some_function(argument):
    '''
        Summary or Description of the Function

        Arguments:
            argument (argument_type): Description of argument

        Returns:
            return_type: Description of return value

    '''

    return return
```

Bugs

- In computing, a bug is an error in code which causes a program to not run as intended.
- Debugging strategies include:
  - Running test cases and comparing the actual result with the expected result
    - The inputs for your test cases should cover the categories of
      - Normal Data (inputs that should be accepted)
      - Boundary/ Extreme Data (inputs that are on the upper and lower boundaries of what should be accepted
      - Error Data (inputs that should be rejected)
  - Using diagnostic `print()` statements in parts of your code to check the value of variables during execution

Errors

- **Syntax Errors:**
  - Errors that are generated due to typing mistakes in the source code
    Eg.
    - Not having a `:` symbol at the end of an `if` statement
    - Not closing off brackets correctly
  - A program will **not** compile unless the syntax errors have been corrected
- **Logic Errors:**
  - Errors in the logic of your program
  - The code will compile without a problem, but the actual result may differ from the result that the programmer expects

Errors

- **Run-time Errors:**
    - Errors that occur during execution and inevitably cause our programs to crash
      Eg.
        - `ZeroDivisionErrors`
        - `IndexErrors`