THE UNIVERSITY OF MELBOURNE

COMP10001

Foundations of Computing

Semester 1, 2021

Tutorial 6

Andrew Naughton

andrew.naughton@unimelb.edu.au

THE UNIVERSITY OF MELBOURNE

Outline

- Documentation
- Commenting Guidelines
- Naming Conventions
- Magic Numbers
- Docstrings
- Bugs
- Errors
- Exercises

THE UNIVERSITY OF MELBOURNE

Documentation

- Adding documentation to our code is a means of communicating the logic behind our approach
- Why 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
  - Helps the author of the code remember what their train of thought was at the time of writing the program

THE UNIVERSITY OF MELBOURNE

Documentation

- We document our code by using:
  - Comments
    - Single line comments start with #
    - Multi-line comments are defined with opening and closing """ or '''
  - Docstrings for functions
  - Meaningful names for functions and variables

THE UNIVERSITY OF MELBOURNE

Commenting Guidelines

- Comments should be concise
- Comments must appear before the relevant block of code

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

- No need to write comments for code that adequately explains itself

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

Commenting Guidelines

- If a bit is happening inside a loop, it can help to use comments to briefly explain what our code does at each distinct logical step

```
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
```

THE UNIVERSITY OF MELBOURNE

Commenting Guidelines

- We typically avoid using *inline* comments (comments on the same line as the code)
- Unless we are describing the purpose of a variable or constant whose use might not be immediately clear

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

THE UNIVERSITY OF MELBOURNE

Naming Conventions

- `snake_case` vs. `camelCase` vs. `PascalCase`
- Python programmers conventionally use the `snake_case` format when naming functions and variables
- For classes (not formally covered in this subject), we use `PascalCase`
- When naming functions and variables, they
  - Must start with a letter or underscore
  - Should be in lowercase
  - Can have numbers
  - Can't be the same as a Python `keyword`

THE UNIVERSITY OF MELBOURNE

Naming Conventions

- As a guide:
  - When naming variables, think of nouns. E.g.
    - `first_name`, `age`, `address`
  - When naming functions, think of verbs (as functions carry out actions). E.g.
    - `get_address(employee_id)`
    - `are_related(person_a, person_b)`
- Single character names are only acceptable in two cases:
  - `for` loop variables that work with the `range`() function
  - The variable refers to a variable in a mathematical equation

Magic Numbers

- Constants which are written into code as literals
- Why are they bad?
  - It's hard to understand their meaning. E.g. `if mark >= 80:`
    - What does `80` represent? H1 cut-off? Pass mark?
  - If the same magic number is used in multiple places in the program, it will make maintenance difficult
    - If we need to change the value, we need to change it manually in all those places
    - Forgetting to change it in one place can lead to incorrect results, hence code becomes error-prone

THE UNIVERSITY OF MELBOURNE

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

- By convention, constants must be uppercase. E.g. `PI = 3.1415`
- Note:
  - Constants must keep the same value for the entirety of the program, i.e. they are “constant” and not “variable”

Docstrings

- A multi-line comment that describes the purpose of a function and provides information on how to use it
- Can be accessed by calling the help() function with your function's name as the argument
- Needed to improve the readability of our code as it provides other users with the information they need to interact with our function, without needing to inspect the source code
- Three components:
  - Short description of purpose
  - The input arguments, their types, and what they represent
  - What will be returned (if anything) 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 argument
```

Bugs

- In computing, a bug is an error in code which causes a program to not run as intended
- Debugging the act of the fixing the bug(s) in a program. Strategies include:
  - Running test cases and comparing the actual output with the expected output
    - 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 our code. E.g.
    - Not having a `:` symbol at the end of an `if` statement
    - Not closing off brackets correctly
  - A program will not compile until all syntax errors have been addressed
- Run-time errors:
  - Errors that occur during execution and inevitably cause our programs to crash. E.g. `ZeroDivisionError`, `IndexError`

THE UNIVERSITY OF MELBOURNE

Errors

- Logic errors:
  - Errors in the logic of our program
  - The code will compile and execute without a problem, but the actual result may differ to the result that the programmer expects

- The three errors types can be distinguished by where they crop up:
  - Compilation: syntax error
  - Execution: run-time error
  - Post-execution: logic error

THE UNIVERSITY OF
MELBOURNE

Exercises