ENGG1811: Computing For Engineers

2025 Term 3

Style, Debugging & Libraries

Week 2: Monday 22nd September, 2025

Monday 14:00 - 16:00 | HarpM15570

Today

- Reminders
- Style Outline
- Conventions Around Variable Names
- Commenting
- Formatting
- Doc-Strings
- Math Library

Reminders

Marking

- Marking starts this lab! You must understand and explain your work.
- If not marked in the lab:
  - Next week's lab if I allow it (usually one free chance or if we ran out of time).
  - If you miss a lab, contact me. You may attend another lab that week or request an extension.
  - For absences longer than one week (week 6 excluded), or if I do not grant an extension, apply for special consideration.
- Wait until after slides/Kahoot to get marked — Kahoot winner skips the queue!
- Complete the MCQ before the lab ends.

Remember to reach out if you need help!

- Learning support page on the course website here: PASS, Help sessions, forum, and Live Coding Sessions. Times and details are listed there.
- Is everyone receiving the lab emails?
- See the lab GitHub repository for these slides and code.
- For questions about labs, assignments, or content, ask in labs or email me at b.mabbutt@student.unsw.edu.au.
- For general issues (e.g. self-directed labs or setup), contact the course account at en1811@cse.unsw.edu.au.
- If a situation out of your control affects your performance, apply for special consideration and keep us informed.

Style Outline

More than just a calculator

- Last week we used Spyder like a simple calculator.
- Now, use the LHS panel to create *scripts* :
  - Save results
  - Reuse calculations
  - Print outputs

Goal For This Lab

- Today we'll practice writing Python scripts, with a focus on style — making code clear and following conventions.
- To get marked off:
  - Write either the revolutions or projectile motion script with near *perfect* style.
  - This is only for today's lab. Style won't be marked in future labs, but assignments will, and I will check it carefully today.

Why should we care about style?

- We want our work to be presentable so others can read and interpret it with minimal effort.
- Certain *conventions* have developed over decades and are widely accepted by programmers.
  - These are not strict rules — your programs rarely fail if you don't follow them exactly.

The Ultimate Source of Truth and Information

- The style guide, found on the course website, contains everything you need to know about style.
- The rest of these slides provides additional justification and explanations for some points, but the main source is the style guide.
- The style guide will be the marking criteria for style in both assignments.

Conventions Around Variable Names

Classifying Data

- Values we assign names to fall into two categories:
  - **Variables:** Values that can change within the program without altering other parts.
  - **Constants:** Values that should not change — altering them would fundamentally change the program or reality.

Examples of Variables & Constants

Questions:

1. In a program calculating a circle's area, is the `radius` a variable or constant?
2. For calculating disk revolutions, is the `number of degrees` in one `revolution` a variable or constant?
3. For calculating business profit, is the `gross income` a variable or constant?
4. For time dilation in a `vacuum`, is the `speed of light` a variable or constant?
5. In a financial program, is the `price of houses` in Sydney a variable or constant?

Convention for Variable Names

- Variables: lowercase with underscores:
  - `profit = -200`
  - `projectile_speed = 3.6`
  - `time = 0.003`
- Constants: uppercase with underscores:
  - `GRAVITY = 9.8`
  - `LIGHT_SPEED = 3 * 10 ** 8`
  - `ANGLES_FULL_CIRCLE = 360`

Descriptive Names

- **Question:** Can we shorten `GRAVITY = 9.8` to `G = 9.8`? Why or why not?

Descriptive Names (Cont)

- We **always** use clear, descriptive names where reasonable.
- Conciseness matters, but clarity is key. Compare:
  - `GRAVITY = 9.8` (preferred)
  - `ACCELERATION_DUE_TO_GRAVITY_NEAR_SURFACE = 9.8` (avoid)
- **Question:** How can we communicate additional context (e.g., valid near Earth's surface) without verbose names?

Commenting

Comments

- The best way to provide clarification and explanation in code is with comments.
- These are placed *above* the code we want to clarify or explain further when it is not *immediately* obvious otherwise.
- Examples:
  - ```
    # assumes we are at the surface of the Earth
    GRAVITY = 9.8
    EARTH_RADIUS = 6.378 * 10 ** 6
    ```
  - ```
    # finds fuel cost for distance travelled
    distance_travelled = speed * total_time
    cost_total = distance_travelled * fuel_price
    ```
  - ```
    MERIDIEM = 12

    # converts 24-hr time to 12-hr time
    regular_time = military_time % MERIDIEM
    ```
    ```

Some Comments on Comments

- Comments do not need to be on every line.
- Keep them short and to the point.
- Provide important info that might be missed: creative choices, assumptions, ignored cases, etc.
- Write *more* comments than *fewer*, but not as many as code.
- Place comments only above complicated lines or blocks of code.

Formatting

Maximum Line Length

- In Spyder, a grey vertical line shows when a line reaches 79 characters.
- Slightly over (≈ 5 characters) is fine; for longer lines, use the `\` to continue the line. This is not needed for comments or code inside `()` brackets.
- Example:

```
income = profit_from_business_one \
         + profit_from_business_two
```

Spacing

Small details make code easier to read:

- Space near operators:
  - `2 + 3` rather than `2+3`
  - `(2 % 3) // 4` rather than `(2%3)//4`
- Space near equality or commas:
  - `meaning = 42` rather than `meaning=42`
  - `print("Hello", "World!")`

Spacing (Cont)

- Space between different sections of code:
  - ```
    # Do this
    length = 3
    height = 4

    area = length * height
    print(area)
    ```
  - ```
    # Don't do this
    length = 3
    height = 4
    area = length * height
    print(area)
    ```

Spacing (Cont II)

- Avoid excessive vertical or horizontal spacing:
  - `# Don't do this either`

```
length  =  3

height  =  4

area = length  *  height

print(area)
```

Doc-Strings

Why Bother With Documentation?

- Programs can get long and convoluted.
- Someone may want to analyze a part of your script or understand the gist quickly.
- Solution: create a doc-string.

Doc-String Structure

1. **Purpose:** 1–2 line overview of what the program is meant to do.
2. **Author:** Your name and anyone you wish to credit.
3. **Date:** Date last revised.
4. **Data Dictionary:** Contains all important variables *and* constants, including their type and purpose.
5. **Method:** Explain how the program achieves its purpose.

Doc-String Pointers

- Purpose, author, and date are usually fine for students; data dictionary and method are often challenging on first attempts.
- **Data Dictionary:**
  - Include name, data type, and short description for each variable/constant. An example is on a slide coming up.
- **Method:**
  - Write in full sentences at a high level.
  - Explain the main idea or process used to satisfy the program's requirements without including technical code details.
  - Provide enough explanation so someone could recreate the program from the method.

Data Types

- We begin with three main data types and will explore more as the course progresses.
  - Integers (`int`): Integers in the mathematical sense.
    - `5`
    - `0`
    - `-4`
  - Floats (`float`): Floating-point numbers (decimals).
    - `2.99792458`
    - `0.0`
    - `9.999999`
  - Strings (`str`): Text of characters.
    - `"Hello"`
    - `"1.2"`
    - `"P@3sw0R|)"`

Data-Dictionary Example

- Consider the following program:

```
# data of a tennis ball's flight
distance = 1000.4
time = 3

speed = distance / time
```

- **Questions:**
  - What is the data-type of distance, time and speed?
  - What could be a description for distance, time and speed?

Complete Doc-String Example

At the *top* of your program:

```
"""
Purpose: To calculate the average speed of a tennis ball
         during its flight.

Author:  Brendan Mabbutt
Date:    18/10/24
Method:  Define the distance and time variables.
         Then calculate the speed by taking the quoteint of
         distance over time and assign it to speed.

Data dictionary:
 [float] distance    The distance in meters the tennis ball
                     has traveled during its time of flight
 [int] time          The tennis ball's time of flight
                     in seconds
 [float] speed       The average speed of the tennis ball
                     while airborne in meters per second
"""
```

Common Errors

It is best to tackle each code error as it occurs. You will improve as the term progresses, so we won't overwhelm you with techniques you might forget. Some common errors to watch for:

- Using a variable before assigning it
- Incorrect usage of the assignment operator ('=')

```
# wrong
x + 1 = x
30 = speed
```

- Leaving out parentheses
- Forgetting that variables are case-sensitive

Math Library

Why do we need to import?

- Base Python is quite skeletal.
  - It does not come with many useful features like certain math operations or the ability to create graphs or tables.
  - **Question:** Why do you think that is? Shouldn't Python include these by default?

Importing Math

- We add functionality by importing a module/library with specific functions — we just need to know its name.
- Example:

```
import math
print(math.sin(30) ** 2 + math.cos(30) ** 2)
```

  - Note: Python needs to look inside `math` ; otherwise, it won't find $\sin$ or $\cos$ in base Python. For example:

    ```
    import math
    print(sin(30) ** 2 + cos(30) ** 2)
    ```

    will NOT work!
  - You can shorten the import for convenience:

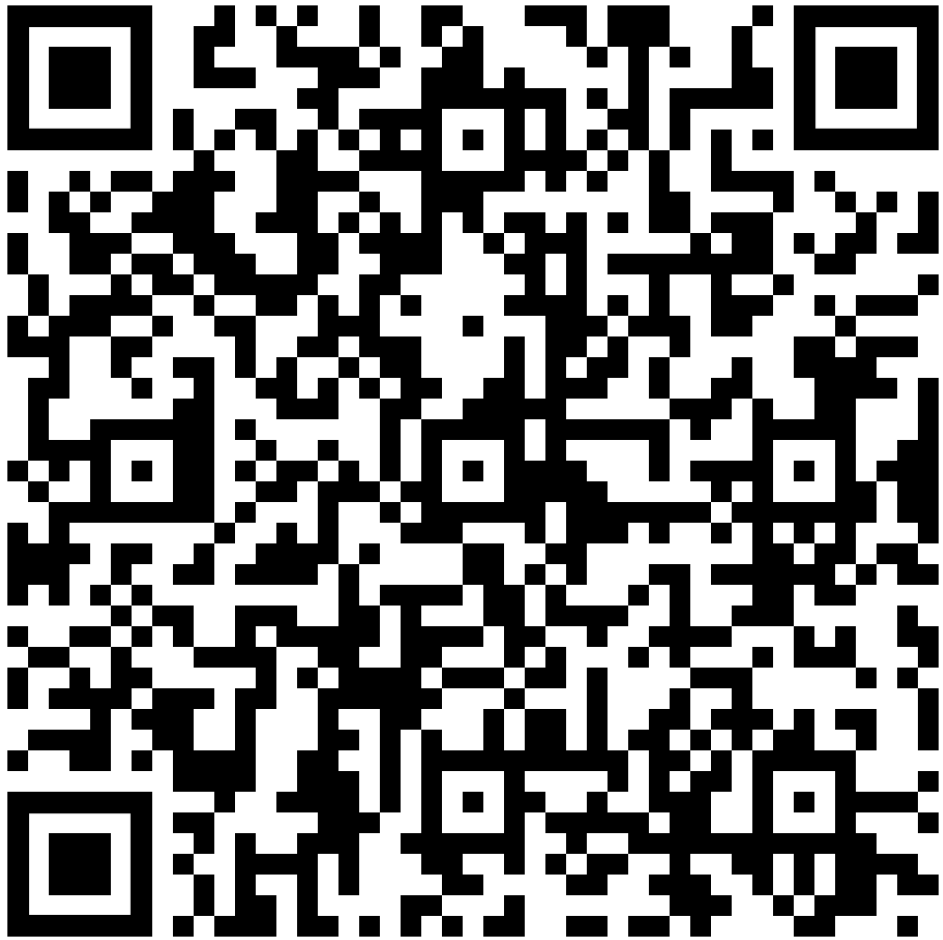
    ```
    import math as m
    print(m.sin(30) ** 2 + m.cos(30) ** 2)
    ```

Lab Tips

- Exercise 1:
  - The error is the same as one of the examples we went through.
- Exercise 2:
  - Remember the convention for defining constants and variables.
  - Part of this exercise is similar to last week's Exercise 2.
- Exercise 3:
  - Read the equations carefully.
  - Use `*` for every instance of multiplication.
- Again: use good coding style for Exercises 1 and 2; see the style guide and ask for help if needed.

Feedback

Feel free to provide anonymous feedback about the lab!



Feedback Form