

# ENGG1811: Computing For Engineers

2025 Term 2

Style, Debugging & Libraries

Week 2: Friday 13<sup>th</sup> June, 2025

Friday 16:00 - 18:00 | OboeME304

# Today

Reminders

Style Outline

Conventions Around Variable Names

Commenting

Formatting

Doc-Strings

Debugging

Math Library

Lab Tips

## Reminders

# Marking

- ▶ Marking starts this lab!
- ▶ If you weren't able to get your work marked off in a lab, you can:
  - ▶ Get your work marked off during the following week's lab if time permits
  - ▶ If you can't attend a lab, contact me . You can attend another lab in the same week to get marked off if you are available
  - ▶ In these cases we will mark you as a lower priority to students needing the current week's lab marked unless it is your first instance
- ▶ If you have completed the lab work before the start of class, you still have to wait until after I go through the slides before you get marked
- ▶ Do not forget to do the MCQ before the end of class!

## Remember to reach out if you need help!

- ▶ Is everyone getting emails for this lab?
- ▶ If you have any questions you would like to ask about labs, assignments, or content, you can ask in the labs or email me at [b.mabbutt@student.unsw.edu.au](mailto:b.mabbutt@student.unsw.edu.au).
- ▶ If you have more **general issues** with this course such as for the self-directed labs or setup, contact the course account at [en1811@cse.unsw.edu.au](mailto:en1811@cse.unsw.edu.au).
- ▶ If there arises a **situation out of your control** affecting your performance in this course, consider applying for special consideration through UNSW. Feel free to loop us in about this.
- ▶ You can also get help in **PASS, Help sessions**, or the **Live Coding Sessions**. Times and details are on the course website. Help sessions will begin in Week 4.

## Style Outline

## Demo: More than just a calculator

- ▶ Last week we treated Spyder as a big, clumsy calculator
- ▶ Hopefully everyone has kept up to date with the lectures, and knows how to now utilise the panel on the LHS of Spyder to create *scrips*
  - ▶ Allows us to save results
  - ▶ Reuse calculations
  - ▶ Print results

# Goal For Today

## Goal

Today, our lab will focus on **style** — making sure our code looks presentable and follows agreed upon conventions.

- ▶ **To get marked off today:**

- ▶ You must create one of your programs — either the revolutions script or the projectile motion script — with near *perfect* style
- ▶ This is only meant for today's lab, all labs after this we will not explicitly be marking for style—the **assignments will** though. I will be marking the style components of all your assignments!



# Why should we care about style?

- ▶ We want to make our work *presentable* so that others can easily read and interpret our work with minimal effort
- ▶ This means that certain *conventions* have developed over the decades that are accepted by most programmers
  - ▶ This is not a hard and fast rule — rarely will your programs halt because you fail to adhere to them

# The Four Main Elements of Style

## 1. Conventions regarding variable names

- ▶ Distinguishing between *variables* and *constants*
- ▶ Providing names which are *descriptive*

## 2. Commenting

- ▶ When to make a comment
- ▶ Exactly why we are making comments in the first place

## 3. Formatting

- ▶ Making sure code looks *aesthetically* clean

## 4. Doc-string

- ▶ This is the most annoying part
- ▶ Most of your effort will be spent here

You must adhere to the above four for perfect style. We will go through more on style as it shows up in the content throughout the course.

# The Ultimate Source of Truth and Information

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

## Conventions Around Variable Names

# Classifying Data

- ▶ Values we assign names to, essentially, fall into two categories:
  - ▶ **Variables:** These are values that, within the context of the program, can be *possibly* changed — you do not need to alter any other detail of your program if you change this variable
  - ▶ **Constants:** These are values that, within the context of the program, *never* makes sense for us to change — there is never a circumstance where these values could be different without fundamentally altering the program or reality

# Examples of Variables & Constants

## Questions:

1. If my program calculates the area of a circle, is the radius of the circle a variable or a constant?
2. If my program wants to calculate the number of times a disk completes a revolution, would the number of degrees in one revolution be a variable or a constant?
3. If my program is to calculate the profit of a business, would the gross income be a variable or a constant?
4. If my program is to calculate the time dilation of a particle in a vacuum, would the speed of light be a variable or a constant?
5. Say I was writing a financial program. Is the price of houses in Sydney a variable or a constant?

# Convention for the Names of Variables

- ▶ For *variables*, our names must be written in all lower case and separated by underscores:
  - ▶ `profit = -200`
  - ▶ `projectile_speed = 3.6`
  - ▶ `time_in_seconds = 0.003`
- ▶ For *constants*, our names must be written in all upper case and separated by underscores:
  - ▶ `GRAVITY = 9.8`
  - ▶ `LIGHT_SPEED = 3 * 10 ** 8`
  - ▶ `ANGLES_IN_FULL_CIRCLE = 360`

# Descriptive Names

- ▶ **Question:** Writing  $\text{GRAVITY} = 9.8$  every time can prove really tiresome. Am I allowed to cut some corners and simply say  $G = 9.8$ ? Why or why not?



# Descriptive Names (Cont)

## Descriptive Names

We will **always** write the slightly longer and descriptive name, where reasonable.

- ▶ This does not mean we place zero value on being concise. Compare the following:
  - ▶ `GRAVITY = 9.8` (Preferred if the meaning is clear)
  - ▶ `ACCELERATION_DUE_TO_GRAVITY_NEAR_SURFACE = 9.8`  
(never do this)
- ▶ **Question:** What if its important to know that this value of gravity is only valid near the surface of the Earth? How do we inform the reader of certain details without making every name verbose?

Commenting

# Comments

- ▶ The better solution to providing clarification and description to code is **comments**.
- ▶ These are placed *above* the code we want to clarify and further explain when it is not *immediately* obvious otherwise
- ▶ Examples:
  - ▶ *# measured at the surface of the Earth*  
GRAVITY = 9.8
  - ▶ *# standard application of formula for speed*  
distance\_travelled = speed \* total\_time
  - ▶ MERIDIEM = 12  
*# converts 24-hr time to 12-hr time but*  
*# discards the AM-PM distinction*  
regular\_time = military\_time % MERIDIEM

# Pointers on Comments

- ▶ Does **not** need to be used for every single line of code!
- ▶ Can be **short and straight** to the point
- ▶ The goal is to give important information that might be lost on the reader: if you do something creatively, assume something, ignore a case, etc.
- ▶ As a general rule of thumb, it is better to write *more* comments than *less*
- ▶ Ideally, comments should be placed **only above complicated lines or blocks** of code.

Formatting

# Maximum Line Length

- ▶ In Spyder, everyone should be able to see a grey vertical line running down your screen
  - ▶ This indicates to you when you have used up 79 characters on any particular line
  - ▶ We have all come to accept that this limit works best for most monitor sizes
- ▶ What if we go past this point?
  - ▶ If it's a little bit over ( $\approx 5$  characters) then this is fine
  - ▶ If it's significantly over, then use the `\`, the break symbol, to inform Python that the current lines continues to the next — does not apply to comments!
- ▶ Example:

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

# Spacing

There are some small details which make our code easier to read

- ▶ Adding space near operators:
  - ▶ `2 + 3` rather than `2+3`
  - ▶ `(2 % 3) // 4` rather than `(2%3)//4`
- ▶ Space near equality or commas:
  - ▶ `meaning = 42` as opposed to `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)

- ▶ Don't be excessive with line spacing or horizontal spacing.

- ▶ *# Don't do this either*

```
length = 3
```

```
height = 4
```

```
area = length * height
```

```
print(area)
```

## Doc-Strings

# Why Bother With Documentation?

- ▶ At a certain point, your programs can get really long and convoluted
  - ▶ What if someone just wants to analyse a certain portion of your script, or wants to just get the general gist?
- ▶ We answer these problems by creating a **doc-string**

# Doc-String Structure

## 1. Purpose

- ▶ A short 1-2 line overview of what this program ultimately does

## 2. Author

- ▶ This is your name, and any person you wish to credit

## 3. Date

- ▶ This should be the date it was last revised

## 4. Data Dictionary

- ▶ This is a list of all important variables and constants

## 5. Method

- ▶ This is the bulk of your doc-string — you will be explaining how your program reaches its purpose and exactly what choices you made along the way.

# Doc-String Pointers

- ▶ Everyone always gets the purpose, author and date correct, but most people trip up with the data dictionary and method
- ▶ Data-dictionary Structure:
  - ▶ Must give the name of every\* variable/constant
  - ▶ Must include the data-type of each variable/constant
  - ▶ Must include a short description of each variable/constant
- ▶ Method Structure:
  - ▶ Must be written in full sentences
  - ▶ Must be written in such a way so that if you only had the method and the rest of the doc-string to work with, you could recreate your entire program with all necessary details

# Data Types

- ▶ We begin with three main data types and will explore more as we progress through the course.
  - ▶ **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:

```
# speed of a tennis ball
```

```
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 quotient 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*

```
"""
```



Debugging

# Common Errors

It is best to tackle each code error as they naturally occur . You will get better at this as the term progresses, so we will not drown you with techniques that you will likely forget to use. In general, however, there are some common errors that you should be on the look out for:

- ▶ Utilising a variable before assigning
- ▶ 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 equipped with many things we might deem useful such as certain math operations, or the ability to create graphs or tables
  - ▶ **Question:** Why do we think that is? Shouldn't Python just include these for us from the get-go?

# Importing Math

- ▶ We give Python additional functionality by importing a module/library with certain functions — all we need to know is the name of said module/library

- ▶ Example:

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

- ▶ Note that we must inform Python to look inside Math, otherwise it will search for `sin` inside its base. For example:

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

will NOT work!

- ▶ You can shorten the import name to make life easier:

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

## Lab Tips

# Lab Tips

- ▶ Exercise 1:
  - ▶ The error is the same as one of the examples we went through
- ▶ Exercise 2:
  - ▶ Make sure you remember the convention for defining constants and variables
  - ▶ Part of this exercise is very similar to last week's Exercise 2
- ▶ Exercise 3:
  - ▶ Read the equations carefully
  - ▶ You need to use `*` for every instance of multiplication

## Feedback

Feel free to provide anonymous feedback about the lab!



Feedback Form