M30299 – Programming
Lecture 05 – Writing High-Quality Code

Matthew Poole & Nadim Bakhshov
moodle.port.ac.uk

School of Computing
University of Portsmouth

2020/21

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 examples from the last few weeks' practicals 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 cofp():
    x = float(input("Enter diameter in cm: "))
    y = 1.5 * math.pi / 4 * x ** 2
    print("The pizza costs", y, "pence")
```

```
def costOfPizza():
    diameter = float(input("Enter diameter in cm: "))
    radius = diameter / 2
    area = math.pi * radius ** 2
    pencePerSquareCm = 1.5
    cost = area * pencePerSquareCm
    print("The pizza costs", 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 _'s;
  - (So `xyz_123` is legal but `1p` and `num 1` are not);
  - it must not be a **keyword** such as `def` or `for`.
- Stick to one of the following styles for names of variables:
  - `mixedCase`, which is recommended since it is used by the graphics module); or
  - `lower_case_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 probably realised 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 blank line(s) 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", pizzasLeft, "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 80 characters as a limit for each line of code.
- 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.,

```
pizzaTotalCost = area * (doughCost + cheeseCost
                         + hamCost + tomatoCost)
sandwichCost = (2 * breadCost) + butterCost  \
               + baconCost + lettuceCost + tomatoCost
```

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 = ((x2 - x1) ** 2 + (y2 - y1) ** 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 costOfPizza():
    diameter = float(input("Diam: ")  # read diameter
    area = math.pi * (diameter / 2) ** 2  # calc area
    cost = area * 1.5  # calculate cost
    print("The cost is", cost)  # display cost
```

Correct code: testing

- Consider the exercise:

  Write a function `euros2pounds` which converts an amount in euros entered by the user to a corresponding amount in pounds. Assume that the exchange rate is 1.10 euros to the pound. *(Hint: be sure to test your solution carefully.)*

- Common incorrect solutions gave one of the following behaviours:

```
Enter amount in euros: 1
The amount in pounds is 1.10
Enter amount in euros: 1.10
The amount in pounds is 1.2100000000000002
```

```
Enter amount in euros: 10
The amount in pounds is 0.11000000000000001
```

Testing your code

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

$$1.10 \text{ euros} = 1 \text{ pound}$$

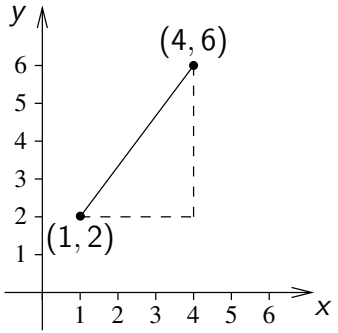
- This fact leads to two things:
  - (after some thought) the correct assignment statement to use in the code (i.e. `pounds = euros / 1.10`); and
  - the first piece of **test data** to use after you've written your function (i.e use 1.10 as input, and expect 1 as the output).
- Notice that you can work out appropriate test data **before** you've written the function (i.e. as you are understanding the task).
- If your function doesn't give the expected output, it's wrong!

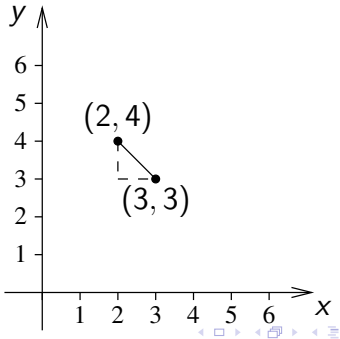
Testing and test data

- 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.

Testing and test data . . . example from worksheet P2

- Write a function `distanceBetweenPoints` that asks the user for four values `x1`, `y1`, `x2` and `y2` that represent two points in two- dimensional space, and then outputs the distance between them.
- A good way to work out test data for this is by plotting points; e.g.





Testing your code . . . examples

- Finally, from practical worksheet 01:

  Write a `futureValue` function that uses a loop to calculate the future value of an investment amount, assuming an annual interest rate of 5.5%. The function should ask the user for the initial amount and the number of years that it is to be invested, and should output the final value of the investment using compound interest with the interest compounded every year.

- What might be some good test data for this exercise?