COMP1730/6730
Programming For Scientists

Overview

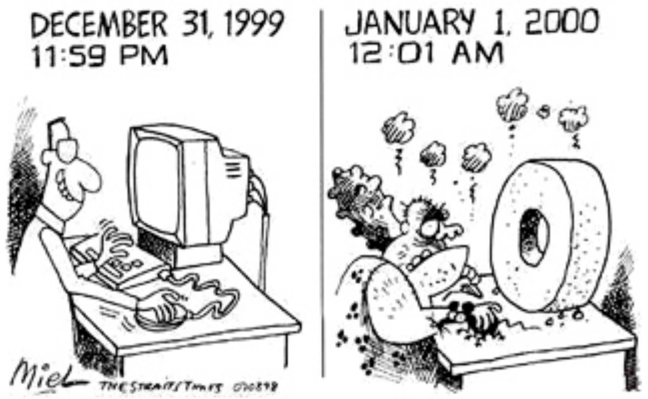
- **Disclaimer!**
    - Software Design is a very big topic: only rough ideas and terminology here
    - For much more detail take COMP2100, COMP2120, and later courses.

- Software Development Methodologies
    - Ideas to implementation: moving between levels of abstraction
- Design Principles
    - Tips and heuristics for making design decisions
- Design Patterns
    - Solving common programming problems
- Conventions and Standards

Why Good Software Design is Important

- **Y2K Problem**
  - Design failure: software using two digits for years (e.g., 99 for 1999)
  - Estimated US$500 billion worldwide to fix

- **Knight Capital "Flash Crash"**
  - Poorly designed (and tested) code switched buying and selling in stock market
  - Company lost $440 million in about 30 minutes



From Idea to Implementation

Cool Idea	• High-level vision
Who? What? How?	• People, things, and interactions
Entities, Structures & Processes	• Modeling
Types, Classes, Functions	• Defining code structure
Code	• Implementation

Two Software Development Methodologies

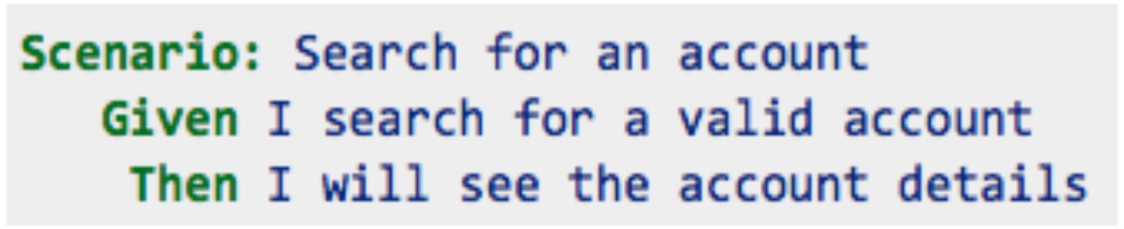
- **Waterfall (“Big Design Up Front”)**
    - Development proceeds in strict sequence:
        - Requirements gathering ⇨ Design ⇨ Implementation ⇨ Verification ⇨ Maintenance
    - Introduced by Royce in 1970 as a non-working model of development
    - Useful for comparison; adopted in some industry

- **Agile Methods**
    - Many variants (e.g., eXtreme Programming, Scrum) with similar philosophy
    - ***Design as an iterative process***
        - Understanding a problem as it is solved
    - Adaptive planning, early delivery, continuous improvement

Design and Development Techniques

- **Use Cases and User Stories**
  - Requirements captured in terms of short stories about how certain features or services might be used
  - Useful for making sure stakeholders and developers agree

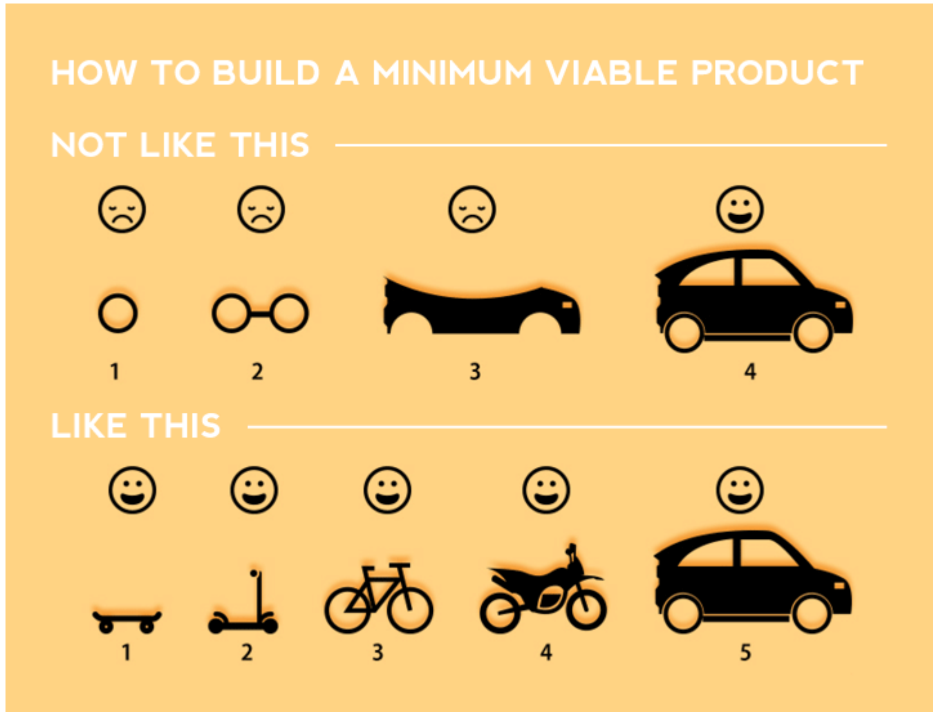
- **Test/Behaviour-Driven Development**
  - Required functionality specified through unit tests and other forms of automated testing
  - Tools exist for systematically turning high-level behavioural specification into executable tests (e.g., Python *behave!*)



```
Scenario: Search for an account
   Given I search for a valid account
    Then I will see the account details
```

Design and Development Techniques

- **“Release Early, Release Often”**
  - Agile methods emphasise having working (but incomplete) code early
  - Enhances feedback from users (Users = You/Group in smaller projects)
- **Failing Fast and Continuous Deployment**
  - Knowing when something is broken as soon as possible means it can be fixed faster
  - Continuously testing code with automated tests and users will find problems early



Some Design Principles

- **Keep it Simple**
  - “Make everything as simple as possible but no simpler”
  - Keep breaking down a problem until what you are trying to do can be explained precisely in a few sentences

- **Separation of Concerns (Modularity)**
  - Reasoning about multiple interactions is difficult
  - Organise aspects of your problem so you can focus on solving one at a time
  - Focus on what information is needed at each stage in a process
  - E.g., HTML + CSS + Javascript; Model, View, Control

Some Design Principles

- **Principle of Least Surprise (Consistency)**
  - Design functions, etc. so that naming, behaviour, arguments, etc. are consistent
  - Stick with familiar conventions whenever possible
- **Don't Repeat Yourself (DRY)**
  - *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."*
  - Less duplicated information means less synchronisation
- **You Ain't Gonna Need It (YAGNI)**
  - *"Always implement things when you actually need them, never when you just foresee that you need them."*
  - Code and features you don't need are breeding grounds for bugs

Design Patterns

- A generic solution to a commonly occurring problem
  - Not a complete design, but a design template

- Examples:
  - **Adapter pattern** allows two incompatible interfaces to work together. Closely related to the **Decorator pattern**, which has a nice implementation in Python.
  - **Iterator pattern** provides a mechanism to traverse every element of a container (without having to know how the elements are stored).
  - **Factory pattern** is used to create objects when the (sub)type is not known until runtime.
  - **Command pattern** is used to prepare a sequence of operations before executing them and can help implement undo features.

Standards and Conventions

- **Standards:** rules for writing and formatting code that are enforced within a project, company or industry.

- **Conventions:** guidelines for how you should write your code so that it is consistent, robust and easily understood by a community of programmers.

"The good thing about standards is that there are so many of them to choose from"
— Grace Hopper

take home message
It doesn't matter which standard you choose as long as you choose, and stick to, one.

Why use Standards?

- Standards help improve readability and ease software maintenance
  - This is especially important when you consider than most software is maintained by someone other than the original author(s)
- Allows the creation of tools to assist with development, documentation, and testing
  - Often makes it easier to find specific functions and search for help
- Eliminates the need to make decisions (*buffet syndrome*)
- Lowers the barrier to learning a new tool or language (prevents lock-in)
  - E.g., standard key bindings for many applications (Ctrl-C and Ctrl-V for cut and paste, resp.)

- Not everyone will like all the conventions used in any given project, but the benefit of consistency that standards bring to a project will outweigh the individual tastes of a single team member
- We have already seen and been using a number of standards in the guise of *good programming practice*.

Mathematical Analogy

- Before standard algebraic notation mathematical theorems used to be stated like this:

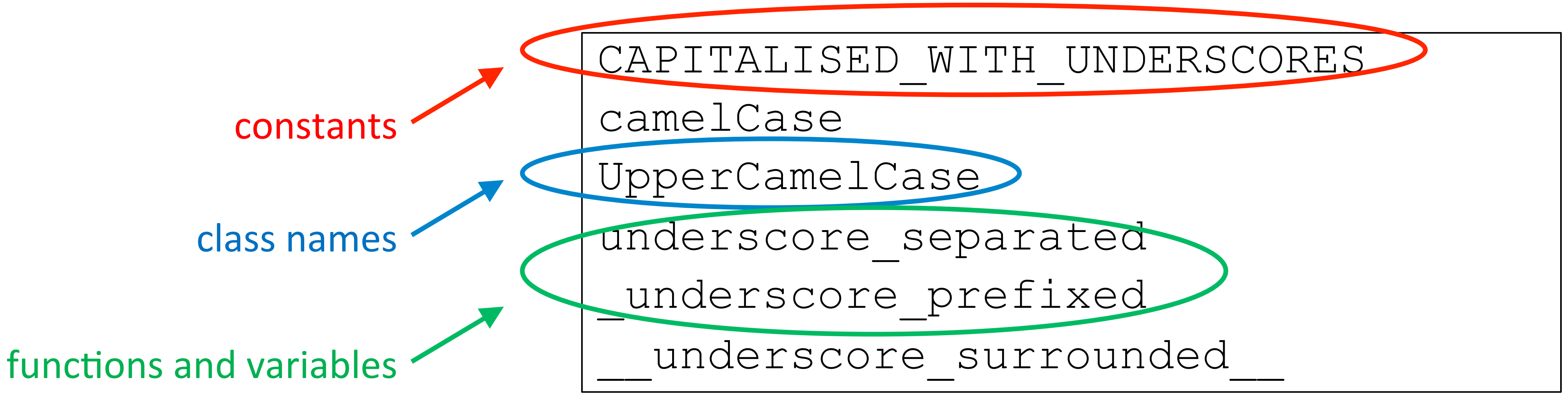
  (Euclid Prop. 47) *In a right triangle the square drawn on the side opposite the right angle is equal to the squares drawn on the sides that make the right angle.*

- With modern notation and conventions we can more simply state Pythagoras's Theorem as (modulo definitions of *a*, *b*, and *c*)

$$a^2 = b^2 + c^2$$

Naming Conventions

- One of the biggest efforts in establishing coding conventions surrounds the naming of variables and functions
- Python restricts names to start with a letter or underscore and contain only letters, numbers and underscores. The rest is up to the programmer.



```
CAPITALISED_WITH_UNDERSCORES
camelCase
UpperCamelCase
underscore_separated
_underscore_prefixed
__underscore_surrounded__
```

Code Layout

- Another important convention has to do with code layout
  - In Python indentation is significant so the language forces a certain layout
  - However, you still have a choice of tab or space, and how many spaces
- The maximum length of any one line of code is also a convention
  - Older programmers used 80 characters, but the new standard is more like 120
  - PyCharm indicates the maximum line limit with a margin line
  - Of course, this is not a hard rule and can sometimes be violated
  - When a line is broken there is a choice about where to break it
    - Often within parentheses, at a comma or inline operation
    - And then how to align the continued line
- Scripts to be wrapped in `if __name__ == "__main__":`

Documentation Conventions

- Use **docstrings** for functions and classes
  - PyCharm helps with automatic docstring templates (type """ <enter>)

```
"""
Concise description of the function’s purpose.

:param first_argument: A short description of the first argument.
:param second_argument: A short description of the second argument.
:return: A short description of the return value.

Optionally, more details about the function including, perhaps, a description of how
it works, what algorithms are implemented, any side effects of the function, special
cases that the caller should be aware of, and examples for how it can be used.
"""
```

- Use **comments** where appropriate (but don’t over do)
  - And always, always keep your comments up to date with the code

Standards for Unit Testing

- There are no hard rules about unit testing but certain conventions are popular
  - Build tests using the `unittest` or `pytest` modules. Similar libraries exist for other programming languages.
  - Have one unit test for each class/module/function in your code.
  - Name unit tests based on what functionality and behaviour is being tested.
  - **Do not** mix test code with production code.
  - agent_tests.py   unit tests for the agent class/module

```
import pytest


def test_get_personality():
```

unit test for the get_personality function

Other Guidelines

- **Brevity.** Keep lambda expressions and list comprehension short (i.e., no more than about one line)
- **Consistency.** When modifying someone else's code follow the existing conventions (even if they don't seem right to you at first)
- **More whitespace.** Actually, use less: avoid extraneous whitespace

Many Python coding conventions are described in **PEP** (Python Enhancement Proposal) **0008 — Style Guide for Python Code** based on the insight that code is read more often that it is written

https://www.python.org/dev/peps/pep-0008/

Tips for Starting a New Project

Questions

- Understand your problem. Can you break it down? Is it similar to other problems you have seen?
- What is the high-level idea? Who are the actors (people, things, interfaces)? How will they be modelled?
- What are your inputs and outputs? How will data flow through your system?
- How will you represent your data? How will you present your results?

Process

- Design top-down. Implement bottom-up. Iterate.
- Set up a repository.
- Implement and test components in isolation. Write test cases.
- Refactor as you go, but don't optimise too early.
- Leave unimportant functionality until later (use dummy functions or `pass`).