

# COMP1531

## 1.2 - SDLC - Intro

# Software Engineering

- What's the difference between **Computer Science** and **Software Engineering**?

# Software Engineering

- What's the difference between **Computer Science** and **Software Engineering**?
  - At UNSW, Software Engineering is an extension of Computer Science, where we give extra focus to how software systems are built, how to manage projects, and how to test software to provide quality assurance.
- Do you need to be a **Software Engineering student** to be employed as a **Software Engineer**?

# Software Engineering

- COMP1511: Learning programming by writing code to solve problems
- COMP1531: Learning Software Engineering by using programming in the context of the software development lifecycle (SDLC)

# Software Engineering

Applying engineering methodologies to our current programming capabilities.

**IEEE definition:** "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software."

We're going to build thought-out, testable, scalable software that is meets set out requirements and is easily maintained

# That was "What" - but "Why"?

What happens in a world without  
good software engineering  
principles being used?

# That was "What" - but "Why"?

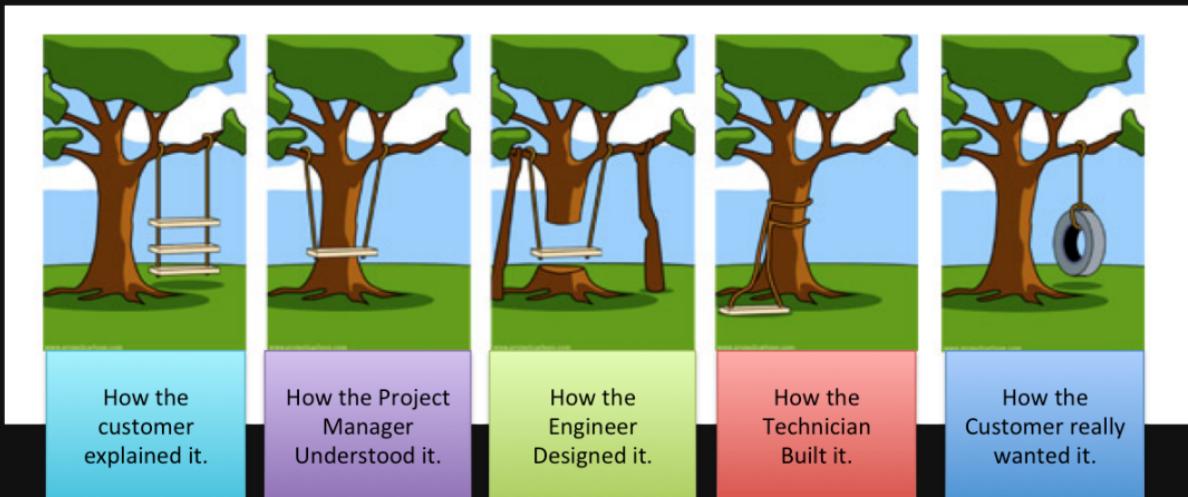
Software engineering fundamentally exists to allow **businesses** and **organisations** to de-risk their business goals compared to just hacking away.

- More predictability about time and budget
- Minimise errors and increase reliability

Software engineering adds small overheads through the software development process to provide higher assurances overall.

Bad things can happen

# That was "What" - but "Why"?



# Software Development Life Cycle (SDLC)



# SDLC

## 1. Requirements Analysis

- Analyse and understand problem domain
- Determine functional and non-functional components
- Generate userstories / use cases

# **SDLC**

## **2. Design**

- Producing software architecture/blue-prints
- System diagrams and schematics
- Modelling of data flows

# **SDLC**

## **3. Development**

- Choose a programming language and write the code

## **4. Testing**

- Use unit or behaviour tests to test your software
- Automating testing for every code change

# **SDLC**

## **5. Deployment**

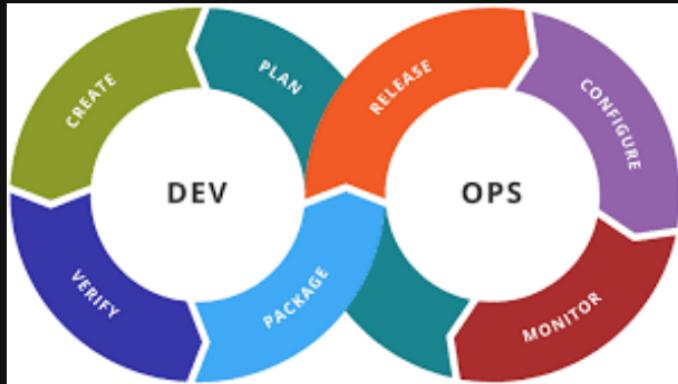
- Make the software available for use by the users

## **6. Maintenance**

- Monitor the system, track issues, interview users to find more requirements

# DevOps

- Modern philosophy of blending the development and operations teams
- Historically was just a merger of "System IT" and "Developers"

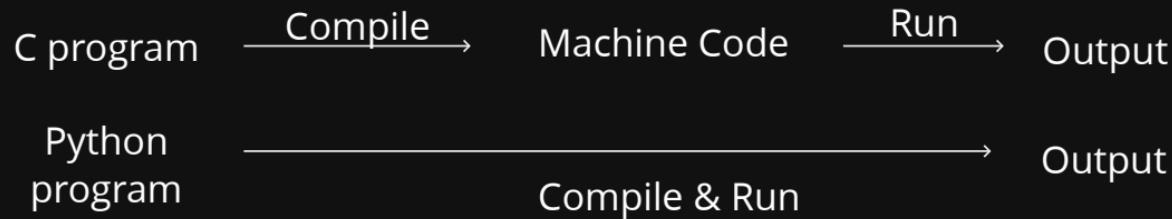


# COMP1531

## 1.4 - Python - Intro

# Python

**Interpreted V compiled**



# Python

Unlike C, Python is a very broad language with a rich range of libraries.

It's simply not possible for us to give you 100 lecture slides to teach you everything you need to know. For many of you, this will be the first course that you have to learn to take the basics from us, and then engage in self-learning to refine your knowledge of the language and its uses.

# Python

## CLI (Command line interface)

- Can be run inline
- Can be run as cli entry
- Can be run via a file

```
hsmith@vx1:~$ python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>>
hsmith@vx1:~$ python3 -c 'print("Hello")'
Hello
hsmith@vx1:~$ █
```

# Python - Basics

## Basic code (basics1.py)

```
1 name = "Giraffe"
2 age = 18
3 height = 2048.11 # mm
4
5 num1 = 3 ** 3
6 num2 = 27 // 3
7
8 print(name + ", " + str(age) + ', ' + str(height))
9 print(name, age, height, sep = ', ')
10 print(f"{name}, {age}, {height}")
11 print(type(name))
12 print(type(age))
13 print(type(height))
14 print(f"3 ** 3 == {num1}")
15 print(f"27 // 3 == {num2}")
```

- Garbage collection
- More info on data types

# Python - Basics

## Strings (**basics2.py**)

```
1 sentence = "My"
2 sentence = sentence + " name is"
3 sentence += " Pikachu"
4
5 print(sentence)
6
7 print("Hi!!" * 10)
```

Python strings are **immutable**

# Python - Basics

## Control structures, argc/argv (basics3.py)

```
1 import sys
2
3 argc = len(sys.argv)
4
5 empty = True
6 if argc > 0:
7     empty = False
8
9 if not empty:
10     if argc == 2:
11         print("Nearly there")
12     elif argc == 3:
13         if sys.argv[1] == "H" and sys.argv[2] == "I":
14             print("HI to you too")
15         else:
16             pass
17     else:
18         print("Please enter two letters as command line")
```

# Python - Basics

## Lists, loops (**basics4.py**)

```
1 names = [ "Hayden", "Rob", "Isaac" ]
2 names.append("Vivian")
3
4 for name in names:
5     print(name)
6
7 print("====")
8
9 names += [ "Eve", "Mia" ]
10 for i in range(0, len(names)):
11     print(names[i])
```

Python lists are very complicated arrays under the hood. You can read a lot [here](#), [here](#), and [here](#).

# Python - Basics

## Tuples (basics5.py)

```
1 x = 5
2 y = 6
3 point = (x, y)
4 print(point)
5
6 a, b = point # destructuring
7 print(f"{a}, {b}")
8
9 names = [ "Giraffe", "Llama", "Penguin" ]
10 for id, name in enumerate(names):
11     print(f"{id} {name}")
```

- lists are mutable, tuples are immutable
- can be modified*      *cannot be modified.*

# Python

python2



python3

# Why Python?

- Rapidly build applications due to high level nature
- Very straightforward toolchain to setup and use
- It's very structured compared to other scripting languages
- Useful in data science and analytics applications

# COMP1531

## 1.5 - SDLC Testing - Intro

# In this lecture

- Basics of pytest (to test code)
- Understanding importing and paths

# C-Style Testing

How did you test in COMP1511?

ctest.c

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 double sum(double a, double b) {
5     return a + b;
6 }
7
8 int main() {
9     assert(sum(1, 2) == 3);
10    assert(sum(2, 2) == 4);
11    assert(sum(3, 2) == 5);
12    printf("All tests passed\n");
13 }
```

# C-Style Testing

Let's first look at python functions

```
1 double sum(double a, double b) {  
2     return a + b;  
3 }
```

```
1 def sum(a, b):  
2     return a + b
```

Q. What are the key differences?

# C-Style Testing

Let's first look at python functions

```
1 double sum(double a, double b) {  
2     return a + b;  
3 }
```

```
1 def sum(a, b):  
2     return a + b
```

Q. What are the key differences?

- No semi-colons
- No braces
- No typing
- "def" to say define function

# C-Style Testing

Q. How would we test this python function?

```
1 def sum(a, b):  
2     return a + b
```

# C-Style Testing

Q. How would we test this python function?

cstyletest.c

```
1 def sum(a, b):
2     return a + b
3
4 assert sum(1, 4) == 3
```

```
:~/teaching/cs1531/19T3-lectures/week1$ python3 cstyletest.py
Traceback (most recent call last):
  File "cstyletest.py", line 4, in <module>
    assert sum(1, 2) == 3
AssertionError
```

# C-Style Testing

Let's clean this up and wrap it in a function,  
though!

```
1 def sum(a, b):  
2     return a + b  
3  
4 def testSmallNumbers():  
5     assert sum(1, 4) == 3  
6  
7 testSmallNumbers()
```

# Basic Python testing

Let's take a look at **pytest**

## What is pytest?

- pytest is a library that helps us write small tests, but can also be used to write larger and more complex tests
- pytest comes with a binary that we run on command line
- pytest detects any **function** prefixed with **test** and runs that function, processing the assertions inside

# pytest - basic

test1\_nopytest.py

```
1 def sum(x, y):
2     return x * y
3
4 def test_sum1():
5     assert sum(1, 2) == 3
6
7 test_sum1()
```

test1\_pytest.py

```
1 import pytest
2
3 def sum(x, y):
4     return x * y
5
6 def test_sum1():
7     assert sum(1, 2) == 3, "1 + 2 == 3"
```

```
1 $ python3 test1_nopytest.py
```

```
1 $ pytest test1_pytest.py
```

# pytest - more complicated

## A more complicated test test\_multiple.py

```
1 import pytest
2
3 def sum(x, y):
4     return x + y
5
6 def test_small():
7     assert sum(1, 2) == 3, "1, 2 == "
8     assert sum(3, 5) == 8, "3, 5 == "
9     assert sum(4, 9) == 13, "4, 9 == "
10
11 def test_small_negative():
12     assert sum(-1, -2) == -3, "-1, -2 == "
13     assert sum(-3, -5) == -8, "-3, -5 == "
14     assert sum(-4, -9) == -13, "-4, -9 == "
15
16 def test_large():
17     assert sum(84*52, 99*76) == 84*52 + 99*76, "84*52, 99*76 == "
18     assert sum(23*98, 68*63) == 23*98 + 68*63, "23*98, 68*63 == "
```

# pytest - prefixes

If you just run

```
$ pytest
```

Without any files, it will automatically look for  
any files in that directory in shape:

- test\_\*.py
- \*\_test.py

# pytest - particular files

You can run specific functions without your test files with the **-k** command. For example, we if want to run the following:

- **test\_small**
- **test\_small\_negative**
- ~~test\_large~~

We could run

**\$ pytest -k small**

or try

**\$ pytest -k small -v**

# pytest - markers

We can also use a range of **decorators** to specify tests in python:

```
1 import pytest
2
3 def pointchange(point, change):
4     x, y = point
5     x += change
6     y += change
7     return (x, y)
8
9 @pytest.fixture
10 def supply_point():
11     return (1, 2)
12
13 @pytest.mark.up
14 def test_1(supply_point):
15     assert pointchange(supply_point, 1) == (2, 3)
16
17 @pytest.mark.up
18 def test_2(supply_point):
19     assert pointchange(supply_point, 5) == (6, 7)
```

```
1 @pytest.mark.up
2 def test_3(supply_point):
3     assert pointchange(supply_point, 100) == (101, 102)
4
5 @pytest.mark.down
6 def test_4(supply_point):
7     assert pointchange(supply_point, -5) == (-4, -3)
8
9 @pytest.mark.skip
10 def test_5(supply_point):
11     assert False == True, "This test is skipped"
12
13 @pytest.mark.xfail
14 def test_6(supply_point):
15     assert False == True, "This test's output is muted"
```

# pytest - more

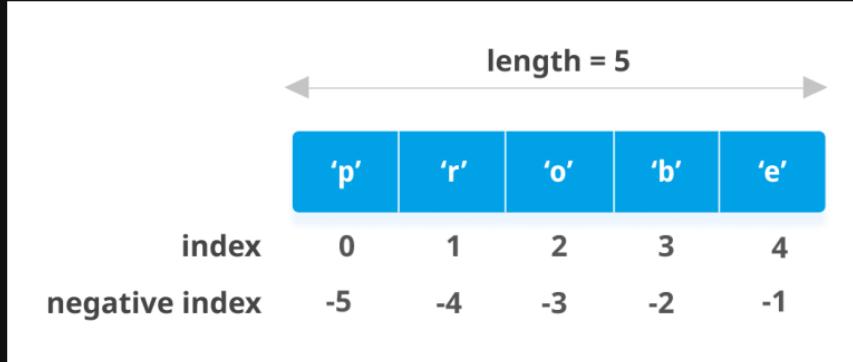
There are a number of tutorials online for pytest.  
This is a very straightforward one.

# COMP1531

## 2.2 - Python - Dictionaries

# Python - Dictionaries

Lists are **sequential containers** of memory. Values are referenced by their **integer index** (key) that represents their location in an **order**



# Python - Dictionaries

**Dictionaries** are **associative containers** of memory. Values are referenced by their **string key** that *maps* to a value

name → "sally"

age → 18

height → "187cm"

# Python - Dictionaries

**Dictionaries** are **associative containers** of memory. Values are referenced by their **string key** that *maps* to a value

*dict\_basic\_1.py*

```
1 userData = {}  
2 userData[ "name" ] = "Sally"  
3 userData[ "age" ] = 18  
4 userData[ "height" ] = "187cm"  
5 print(userData)
```

```
1 {'name': 'Sally', 'age': 18, 'height': '187cm'}
```

# Python - Dictionaries

There are a number of different ways  
we can construct and interact with  
dictionaries

*dict\_basic\_2.py*

```
1 userData = {  
2     'name' : 'Sally',  
3     'age' : 18,  
4     'height' : '186cm', # Why a comma?  
5 }  
6(userData['height']) = '187cm'  
7 print(userData)
```

```
1 {'name': 'Sally', 'age': 18, 'height': '187cm'}
```

# Python - Dictionaries

*dict\_loop.py*

Basic loops are over **keys**  
not **values**:

How would we modify this  
to print out the values  
instead?

```
1 userData = [
2     {
3         'name' : 'Sally',
4         'age' : 18,
5         'height' : '186cm',
6     },
7     {
8         'name' : 'Bob',
9         'age' : 17,
10        'height' : '188cm',
11    },
12]
12 for user in userData:
13     print("Whole user: ", user)
14     for part in user:
15         print(f" {part}")
```

```
1 Whole user: {'name': 'Sally', 'age': 18, 'height': '186cm'}
2     name
3     age
4     height
5 Whole user: {'name': 'Bob', 'age': 17, 'height': '188cm'}
6     name
7     age
8     height
```

# Python - Dictionaries

*dict\_loop\_2.py*

```
1 userData = {'name' : 'Sally', 'age' : 18, \
2             'height' : '186cm'}
3
4 for user in userData.items():
5     print(user)
6 print("====")
7
8 for user in userData.keys():
9     print(user)
10
11 print("====")
12 for user in userData.values():
13     print(user)
```

```
1 ('name', 'Sally')
2 ('age', 18)
3 ('height', '186cm')
4 =====
5 name
6 age
7 height
8 =====
9 Sally
10 18
11 186cm
```

# Python - Dictionaries

## Optional Activity

Q. Write a python program that takes in a series of words from STDIN and outputs the frequency of how often each vowel appears

# COMP1531

## 2.3 - Python - Globals

# Global Variables

- All variables in the python global scope can be accessed (read) in any other scope in the program.
- To modify a global variable you must use the "global" keyword at the top of the function using it.

global.py

```
1 myVariable = 5
2
3 def printSomething():
4     print(myVariable)
5
6 def writeSomethingBad():
7     myVariable = 6
8
9 def writeSomethingGood():
10    global myVariable
11    myVariable = 6
```

# Global Variables

- How will we use this information to make iteration 1 easier?

# COMP1531

## 2.4 - Python - Importing & Paths

# importing and modules

- In python you're able to write code in one file, and import it into another file (just like C).

# importing and modules

calmath.py

```
1 def daysIntoYear(month, day):
2     total = day
3     if month > 0:
4         total += 31
5     if month > 1:
6         total += 28
7     if month > 2:
8         total += 31
9     if month > 3:
10        total += 30
11    if month > 4:
12        total += 31
13    if month > 5:
14        total += 30
15    if month > 6:
16        total += 31
17    if month > 7:
18        total += 30
19    if month > 8:
20        total += 31
21    if month > 9:
22        total += 30
23    if month > 10:
24        total += 31
25    return total
26
27 def quickTest():
28     print(f"month 0, day 0 = {daysIntoYear(0,0)}")
29     print(f"month 11, day 31 = {daysIntoYear(11,31)}")
30
31 #if __name__ == '__main__':
32 #    quickTest()
33
34 quickTest()
```

importto.py

```
1 import sys
2
3 import calmath
4
5 if len(sys.argv) != 3:
6     print("Usage: importto.py month dayofmonth")
7 else:
8     print(calmath.daysIntoYear(int(sys.argv[1]), \
9                               int(sys.argv[2])))
```

What is this for??  
(Live Demo)



# Ways to import

use.py

```
1 import * from lib
2
3 # To do
4
5 from lib import one, two, three
6
7 # To do
8
9 import lib
10
11 # To do
```

lib.py

```
1 def one():
2     return 1
3
4 def two():
5     return 2
6
7 def three():
8     return 3
```

Which ways do we prefer and why?

# pytests and paths

- Pathing with pytest is a big complication. Because pytest is a program, it actually runs many python files in a container, which means that many different python files are run from different folders.
- This makes our assumptions around paths difficult...

# "testpath" example

## Live demo

Let's look at week 2 lecture code to learn more about importing, pytests, and paths

# Python Path

This is something needed to make  
pytest work

If your project is in ~/cs1531/project

```
1 export PYTHONPATH="$PYTHONPATH:~/cs1531/project"
```

You can add this line to your  
~/.bashrc if you don't want to type it  
in every time you open a terminal

# COMP1531

## 2.5 - Python - Packages

# Importing libraries

Python comes packaged with a number of standard libraries (e.g. "math"). However, many libraries that you may want to use have to be installed for usage.

Installing is quite easy due to the **pip** program, which makes installations of python dependencies doable in just a single line of code. **pip** is a package installer for python.

# Importing libraries

To use the `numpy` library we need to first install it on our machine.

```
1 $ pip3 install numpy
```

```
1 import numpy as np
2
3 a = np.array(42)
4 b = np.array([1, 2, 3, 4, 5])
5 c = np.array([[1, 2, 3], [4, 5, 6]])
6 d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
7
8 print(a.ndim)
9 print(b.ndim)
10 print(c.ndim)
11 print(d.ndim)
```

# Virtual Environments

A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them.

You can read more about them [here](#) and [here](#).

You may be asked a question about them on the exam, but you will never be required to use them.

They are often required for use with CI/CD

# Virtual Environments

Even though we know how to install modules, we now run into a problem:

- How do I easily share the modules that I've installed with my team members?
- How do I ensure my project doesn't end up accidentally using installed modules from other projects, and vice versa?

# Virtual Environments

```
1 pip3 install virtualenv
2 python3 -m virtualenv venv/
3 source venv/bin/activate
4
5 # Do stuff
6
7 pip3 freeze > requirements.txt # Save modules
8 pip3 install -r requirements.txt # Install modules
9
10 deactivate
```

# COMP1531

## 2.6 - Python - Exceptions

# Python - Exceptions

An **exception** is an action that disrupts the normal flow of a program. This action is often representative of an error being thrown. Exceptions are ways that we can elegantly recover from errors

# Python - Exceptions

The simplest way to deal with problems...

**Just crash**

*exception\_1.py*

```
1 import sys
2
3 def sqrt(x):
4     if x < 0:
5         sys.stderr.write("Error Input < 0\n")
6         sys.exit(1)
7     return x**0.5
8
9 if __name__ == '__main__':
10    print("Please enter a number: ")
11    inputNum = int(sys.stdin.readline())
12    print(sqrt(inputNum))
```

# Python - Exceptions

Now instead, let's raise an exception

However, this just gives us more information,  
and doesn't help us handle it

*exception\_2.py*

```
1 import sys
2
3 def sqrt(x):
4     if x < 0:
5         raise Exception(f"Error, sqrt input {x} < 0")
6     return x**0.5
7
8 if __name__ == '__main__':
9     print("Please enter a number: ",)
10    inputNum = int(sys.stdin.readline())
11    print(sqrt(inputNum))
```

# Python - Exceptions

If we catch the exception, we can better handle it

*exception\_3.py*

```
1 import sys
2
3 def sqrt(x):
4     if x < 0:
5         raise Exception(f"Error, sqrt input {x} < 0")
6     return x**0.5
7
8 if __name__ == '__main__':
9     try:
10         print("Please enter a number: ")
11         inputNum = int(sys.stdin.readline())
12         print(sqrt(inputNum))
13     except Exception as e:
14         print(f"Error when inputting! {e}. Please try again:")
15         inputNum = int(sys.stdin.readline())
16         print(sqrt(inputNum))
```

# Python - Exceptions

Or we could make this even more robust

*exception\_4.py*

```
1 import sys
2
3 def sqrt(x):
4     if x < 0:
5         raise Exception(f"Error, sqrt input {x} < 0")
6     return x**0.5
7
8 if __name__ == '__main__':
9     print("Please enter a number: ",)
10    while True:
11        try:
12            inputNum = int(sys.stdin.readline())
13            print(sqrt(inputNum))
14            break
15        except Exception as e:
16            print(f"Error when inputting! {e}. Please try again:")
```

# Python - Exceptions

Key points:

- Exceptions carry data
- When exceptions are thrown, normal code execution stops

*throw\_catch.py*

```
1 import sys
2
3 def sqrt(x):
4     if x < 0:
5         raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
6     return x**0.5
7
8 if __name__ == '__main__':
9     if len(sys.argv) == 2:
10         try:
11             print(sqrt(int(sys.argv[1])))
12         except Exception as e:
13             print(f"Got an error: {e}")
```

# Python - Exceptions

Examples with pytest (very important for project)

*pytest\_except\_1.py*

```
1 import pytest
2
3 def sqrt(x):
4     if x < 0:
5         raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
6     return x**0.5
7
8 def test_sqrt_ok():
9     assert sqrt(1) == 1
10    assert sqrt(4) == 2
11    assert sqrt(9) == 3
12    assert sqrt(16) == 4
13
14 def test_sqrt_bad():
15     with pytest.raises(Exception, match=r"*\u00d7Cannot sqrt*"):
16         sqrt(-1)
17         sqrt(-2)
18         sqrt(-3)
19         sqrt(-4)
20         sqrt(-5)
```

# Python - Exception Sub-types

Other basic exceptions can be caught  
with the "Exception" type

*pytest\_except\_2.py*

```
1 import pytest
2
3 def sqrt(x):
4     if x < 0:
5         raise ValueError(f"Input {x} is less than 0. Cannot sqrt a number < 0")
6     return x**0.5
7
8 def test_sqrt_ok():
9     assert sqrt(1) == 1
10    assert sqrt(4) == 2
11    assert sqrt(9) == 3
12    assert sqrt(16) == 4
13
14 def test_sqrt_bad():
15     with pytest.raises(Exception):
16         sqrt(-1)
17         sqrt(-2)
18         sqrt(-3)
19         sqrt(-4)
20         sqrt(-5)
```

# COMP1531

## 2.7 - PM & Teamwork - Intro

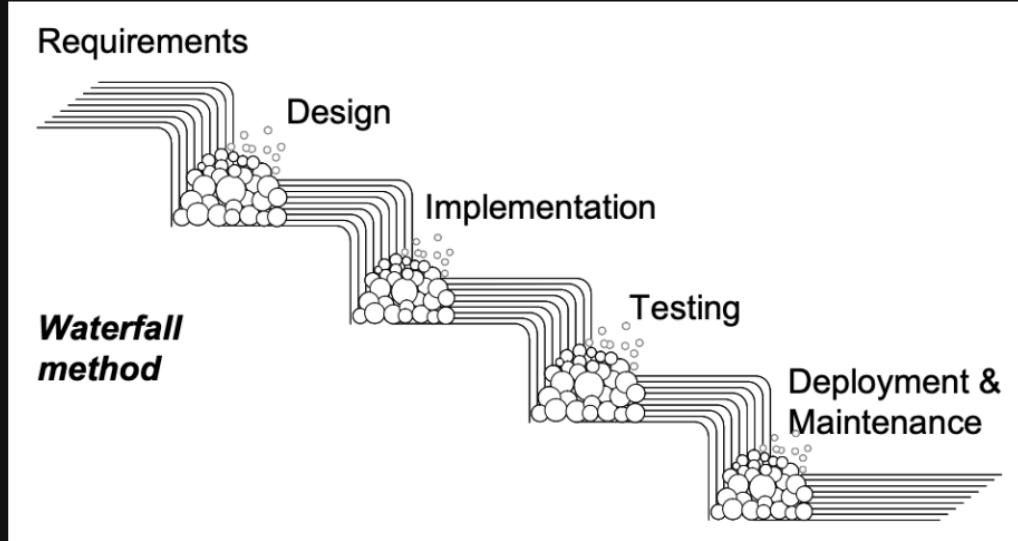
# What is agile?



# Yeah, but what is it really?

- Philosophy
- Practices
- Processes
- A cultural movement?

# A brief history lesson



# History is a lie

- "Waterfall" has never been proposed as a viable software methodology
- Reference:  
<http://www.idinews.com/waterfall.html>

# Defining features (that people usually agree on)

- Iterative and incremental
- Quick turnover
- Light on documentation

# So what is agile good for?

- Your resume?
- Changing requirements
- Delivering software on time
- Your project?

# Agile Practices

- Practices today, processes later on
- We will focus on the ones you will find most helpful in your project

# Standups

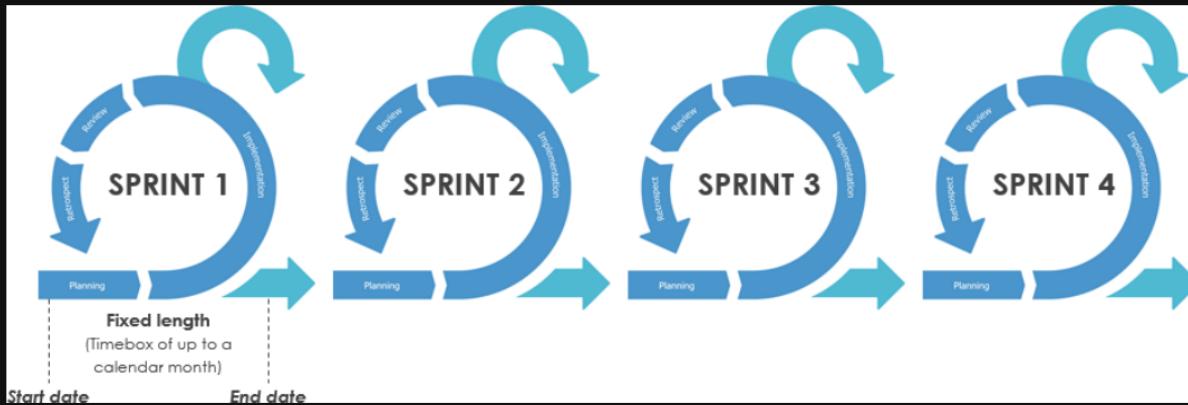
- Frequent (often daily) **short** progress update meetings
- Traditionally, everyone stands up
- Answer 3 key questions
  - What did I do?
  - What problems did I face?
  - What am I going to do?

# Asynchronous Stand-ups

- A somewhat controversial topic
- Advantages
  - No need to find a suitable time for everyone
  - May work better for big teams
- Disadvantages
  - "Blockers" take longer to be addressed
  - Easy to forget to give an update
  - Less personal
  - Updates from others can be missed

# Sprints/iterations

- Time is fixed, scope is flexible
- Plan only for the next sprint
- Typically have a release at the end of each sprint



# Task Boards



# Taskboards

- Available in GitLab
- Use them to store and track your progress on user stories
- You don't need many columns. E.g.
  - Backlog
  - Todo
  - Doing
  - Testing?
  - Closed/Done

# Pair programming

- Two programmers, one computer, one keyboard
- Take it in turns to write code, but discuss it as they go
- Can result in better code quality
- Good for helping less experienced programmers learn *micro-techniques* from more experienced programmers

# Test-Driven-Development (TDD)

- Writing tests *before* the implementation
- Write only enough code to make the next test pass
- Takes some practice
- We'll come back to this next week

# Meeting Minutes

- For more formal meetings (e.g. weekly meetings) it's quite common to take *minutes* of the meeting.
- Meeting minutes will typically consist of documenting:
  - Attendees
  - (Optional) Agenda
  - Discussion Points
  - Actions

Date	Attendees	Agenda	Notes, decisions and action items
Jan 11, 2021	@Ana [REDACTED] [REDACTED] @Anne [REDACTED] [REDACTED] @Hayden Smith	<input checked="" type="checkbox"/> @Ana [REDACTED] to have joint/trust data from Kurt so that @Hayden Smith @Anne [REDACTED] can make decision & start the plan <input checked="" type="checkbox"/> Update on Q2 initiative(s) <input checked="" type="checkbox"/> @Anne [REDACTED] to quickly share Invest drafts	<ul style="list-style-type: none"><li>• @Hayden Smith To do a 30 minute run through the mobile site for feedback</li><li>• @Hayden Smith Post in team-leads about multi-brokerage</li><li>• @Hayden Smith Make portfolio/profile tickets</li><li>• @Anne [REDACTED] to work on profile/portfolio integration</li><li>• @Hayden Smith to make confluence page for "template portfolios"</li><li>• @Hayden Smith to push to prod every day</li><li>• @Hayden Smith Reach out to DW for brokerage</li></ul>

# COMP1531

## 3.1 - Python - Objects

# What are objects?

# Objects

- Technically, a fairly simple idea
- Conceptually, a rich area of software design with complicated outcomes
- There's a whole course on Object-Oriented Design and Programming, so we'll only focus on basic stuff here

Python is not an Object-Oriented language. It's a scripting language with class capabilities.

# A simple example

obj.py

```
1 from datetime import date
2
3 today = date(2019, 9, 26)
4
5 # 'date' is its own type
6 print(type(today))
7
8 # Attributes of 'today'
9 print(today.year)
10 print(today.month)
11 print(today.day)
12
13 # Methods of 'today'
14 print(today.weekday())
15 print(today.ctime())
```

# Objects in python

- Contain *attributes* and *methods*
- Attributes are values inside objects
- Methods are functions inside objects
- Methods can read or modify attributes of the object

# Everything\* is an object

- Almost all values in python are objects
- For example:
  - lists have an `append()` method

```
1 animals = ["dog", "cat", "chicken"]
2 animals.append("sheep") # Modifies the list 'animals'
```

- ▪ strings have a `capitalize()` method

```
1 greeting = "hi there!"
2 print(greeting.capitalize()) # Returns a new string
```

# Creating objects

- *Classes* are blueprints for objects

student.py

```
1 class Student:  
2     def __init__(self, zid, name):  
3         self.zid = zid  
4         self.name = name  
5         self.year = 1  
6  
7     def advance_year(self):  
8         self.year += 1  
9  
10    def email_address(self):  
11        return self.zid + "@unsw.edu.au"  
12  
13 rob = Student("z3254687", "Robert Leonard Clifton-Everest")  
14 hayden = Student("z3418003", "Hayden Smith")
```

# Details

- Methods can be *invoked* in different ways
  - rob.advance\_year()
  - Student.advance\_year(rob)
- The 'self' argument is implicitly assigned the object on which the method is being invoked
- The '\_\_init\_\_()' method is implicitly called when the class is *constructed*

# Managing Data Example

**Activity:** Use the data in  
<https://www.cse.unsw.edu.au/~cs1531/20T3/weatherAUS.csv> to  
write a python program to determine the location with the  
most rain over the last years

# Extra Help

- Fixture example (in week 3 lecture code)
- Python path setting with `pwd`
- These questions have the same answer

# COMP1531

## 3.2 - Python - Pythonic

# Being Pythonic

Being "Pythonic" means that your code generally follows a set of idioms agreed upon by the broader python community.

*"When a veteran Python developer calls portions of code not "Pythonic", they usually mean that these lines of code do not follow the common guidelines and fail to express its intent in what is considered the most readable way. On some border cases, no best way has been agreed upon on how to express an intent in Python code, but these cases are rare."*

**Hitchhiker's guide to python ([read more on this](#))**

# Examples

- Docstrings
- Map, reduce, filter, lambdas
- Exceptions > Early returns
- Destructuring, ignored variables
- Enumerate
- Multi line strings

# Docstrings

Docstrings are an important way to document code and make clear to other programmers the intent and meaning behind what you're writing. We are somewhat different on the formatting, but we want it to include 1) Description, 2) Parameters, 3) Returns

## docstring.py

```
1 def string_find(str1, str2):
2     """ Returns whether str2 can be found within str1
3
4     Parameters:
5         str1 (str): The haystack
6         str2 (str): The needle
7
8     Returns:
9         (bool): Whether or not str2 could be found in str1
10
11    """
12
```

# Map, Reduce, Filter

- **Map**: creates a new list with the results of calling a provided function on every element in the given list
- **Reduce**: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value
- **Filter**: creates a new array with all elements that pass the test implemented by the provided function

# Map

**Map**: creates a new array with the results of calling a provided function on every element in the calling array

## map.py

```
1 def shout(string):
2     return string.upper() + "!!!!"
3
4 if __name__ == '__main__':
5     tutors = ['Simon', 'Teresa', 'Kaiqi', 'Michelle']
6     angry_tutors = list(map(shout, tutors))
7     print(angry_tutors)
```

# Reduce

**Reduce**: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value

## reduce.py

```
1 from functools import reduce
2
3 def custom_sum(first, second):
4     return first + second
5
6 if __name__ == '__main__':
7     studentMarks = [ 55, 43, 34, 23, 22, 10, 44 ]
8     total = reduce(lambda a, b: a + b, studentMarks)
9     print(total)
```

# Filter

**Filter:** creates a new array with all elements  
that pass the test implemented by the  
provided function

## filter.py

```
1 from functools import reduce
2
3 if __name__ == '__main__':
4     marks = [ 65, 72, 81, 40, 56 ]
5     passing_marks = list(filter(lambda m: m >= 50, marks))
6     total = reduce(lambda a, b: a + b, passing_marks)
7     average = total/len(passing_marks)
8     print(average)
```

# Combined

## allthree.py

```
1 from functools import reduce
2
3 if __name__ == '__main__':
4     marks = [ 39, 43.2, 48.6, 24, 33.6 ] # Marks out of 60
5     normalised_marks = map(lambda m: 100*m/60, marks)
6     passing_marks = list(filter(lambda m: m >= 50, normalised_marks))
7     total = reduce(lambda a, b: a + b, passing_marks)
8     average = total/len(passing_marks)
9     print(average)
```

# Exceptions > Early Returns

You might be quite familiar with early returns:

**early.py**

```
1 def sqrt(num):
2     if num < 0:
3         return None
4     return num ** 0.5
5
6 myNum = int(input())
7 if sqrt(myNum) is not None:
8     print(sqrt(myNum))
```

The problems though are:

- Often we can only use "None" or some arbitrary return (-1) to signify that it didn't work
- It's harder to check for a client using it

# Exceptions > Early Returns

So we use exceptions. And we can make our own.

## early.py

```
1 class SqrtException(Exception):
2     pass
3
4 def sqrt(num):
5     if num < 0:
6         raise SqrtException("Number cannot be < 0")
7     return num ** 0.5
8
9 try:
10     print(sqrt(int(input())))
11 except SqrtException as e:
12     print(e)
```

# Destructuring

Being able to make tuples and destructure them is very powerful. If you don't want all tuples you can use blanks to ignore them.

## destructure.py

```
1 import math
2
3 def convert(x, y):
4     return (math.sqrt(x**2 + y**2), math.degrees(math.atan2(y,x)))
5
6 if __name__ == '__main__':
7     print("Enter x coord: ", end=' ')
8     x = int(input())
9     print("Enter y coord: ", end=' ')
10    y = int(input())
11
12    mag, dir = convert(x, y)
13    print(mag, dir)
14
15    mag2, _ = convert(x, y)
16    print(mag2)
```

# Enumerate

Sometimes we want to iterate cleanly over the values in a list, but also know what index we're up to. In these situations the enumerate built-in is useful.

## enumerate.py

```
1 tutors = ['Vivian', 'Rob', 'Rudra', 'Michelle']
2
3 for idx, name in enumerate(tutors):
4     print(f"{idx + 1}: {name}")
```

# Multi-line strings

Someones strings need to exist over multiple lines,  
there are two good approaches for this

## multiline.py

```
1 if __name__ == '__main__':
2     text1 = """hi
3
4     this has lots of space
5
6     between chunks"""
7
8     text2 = (
9         "This is how you can break strings "
10        "into multiple lines "
11        "without needing to combine them manually"
12    )
13
14     print(text1)
15     print(text2)
```

# COMP1531

## 3.3 - SDLC Testing - pylint

What is good style?

Programs must be written for  
people to read, and only  
incidentally for machines to  
execute – *Abelson & Sussman*,  
*"Structure and Interpretation of  
Computer Programs"*

# Style

- Ultimately about readability and maintainability
- Style guides give rules of thumb and conventions to follow
- ...but good style is ultimately hard, if not impossible, to measure
- That said, tools can be a lot of help

There are a lot of  
tools in modern  
software  
engineering

# Pylint

- An external tool for *statically* analysing python code
- Can detect errors, warn of potential errors, check against conventions, and give possible refactorings
- By default, it is **very** strict
- Can be configured to be more lenient

# Controlling Messages

- Disable messages via the command line

```
$ pylint --disable=<checks> <files_to_check>
```

```
$ pylint --disable=missing-docstring <files>
```

- Disable messages in code; e.g.

```
if year % 4 != 0: #pylint: disable=no-else-return
```

- Disable messages via a config file

- If a .pylintrc file is in the current directory it will be used

- Can generate one with:

```
pylint <options> --generate-rcfile > .pylintrc
```

# **COMP1531**

## **3.4 - SDLC Testing - Verification & Validation**

# Verification

Verification in a system life cycle context is a set of activities that compares a product of the system life cycle against the required characteristics for that product. This may include, but is not limited to, specified requirements, design description and the system itself.

# Validation

Validation in a system life cycle context is a set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals and objectives.

\*ISO/IEC/IEEE 29148:2018

# Verification

System has been built right

# Validation

The right system has been built

\*by intuition

# Formal Verification

- Proving (via Mathematics) that a piece of software has certain desirable properties
- Treats the software, or the algorithms implemented in the software, as a mathematical object that can be reasoned about.
- Typically involves tools like proof assistants, model checkers or automatic theorem provers.
- **Not something we cover in this course**

# Formal Verification

- Tends to have a high cost in terms of effort
- E.g. to verify a microkernel
  - it took ~20 person years
  - and ~480,000 lines of proof script
  - for ~10,000 of C

What is testing  
anyway?

“Testing shows the  
presence, not the  
absence of  
bugs” – *Edsger W.  
Dijkstra*

# Unit testing

ISTQB definition:

The testing of individual software components

Method:

White-box

Who:

Software Engineers

# Integration Testing

ISTQB definition:

Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

Method:

White-box or Black-box

Who:

Software Engineers or independent testers

# System Testing

ISTQB definition:

The process of testing an integrated system to verify that it meets specified requirements.

Method:

Black-box

Who:

Normally, independent testers

# Acceptance Testing

ISTQB definition:

Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

Method:

Black-box

Who:

User or Customer

# COMP1531

## 3.5 - SDLC Testing - Code Coverage

How do we know if  
our tests are good?

# Coverage

- **Test Coverage:** a measure of how much of the feature set is covered with tests
- **Code coverage:** a measure of how much code is executed during testing

# Example: Leap years

```
1 def is_leap_year(year):
2     if year % 4 != 0:
3         return False
4     elif year % 100 != 0:
5         return True
6     elif year % 400 != 0:
7         return False
8     else:
9         return True
```

# Coverage.py

- Measure code coverage as a percentage of statements (lines) executed
- Can give us a good indication how much of our code is executed by the tests
- ... and most importantly highlight what has **not** been executed.

# Example: Year from day

```
1 def day_to_year(days):
2     """
3         Given a number of days from January 1st 1970, return the year.
4     """
5     year = 1970
6
7     while days > 365:
8         if is_leap_year(year):
9             if days > 366:
10                 days -= 366
11                 year += 1
12         else:
13             days -= 365
14             year += 1
15
16     return year
```

# Checking code coverage

- Run Coverage.py for your pytests:

```
coverage run --source=. -m pytest
```

- View the coverage report:

```
coverage report
```

- Generate HTML to see a breakdown (puts report in htmlcov/)

```
coverage html
```

# Case study: Zune Bug

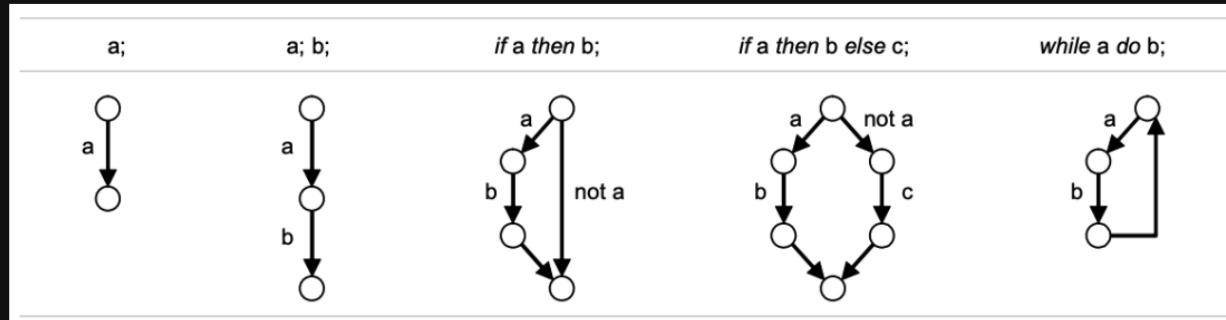
- On December 31st 2008, Microsoft Zunes stopped working for the whole day.
- The bug in the above code caused them to go into an infinite loop
- Hardly catastrophic, but embarrassing for Microsoft



Can we find the Zune bug  
with testing?

# Branch coverage checking

- For lines that can potentially jump to more than one other line (e.g. if statements), check how many of the possible branches were taken during execution
- Can be done with the --branch option in Coverage.py
- Sometimes referred to as edge coverage



Does code coverage imply  
test coverage?

**What is the right  
level of code  
coverage?**

# Summary

- Code coverage is useful
- It's more important to look at what's not covered than the coverage percentage
- Branch coverage is a more accurate measurement so you should use it instead of statement coverage
- Like all measurements, it's important to understand what meaning to attach to it

# COMP1531

## 4.1 - SDLC Development - Data Transfer

# Standard Interfaces

In any field in engineering, we often have systems, components, and designs built by different parties for different purposes.

How do all of these systems connect together?  
Through standard interfaces

# Standard Interfaces



# Data Interchange Formats

When it comes to **transferring data**, we also need common interface(s) that people all send or store data in universal ways to be shared between applications or shared over networks.

Three main interchange formats we will talk about:

- JSON
- YAML
- XML

# JSON

*JavaScript Object Notation (JSON) - TFC 7159*

A format made up of braces for dictionaries, square brackets for lists, where all non-numeric items must be wrapped in quotations. Very similar to python data structures.

# JSON

Let's represent a structure that contains a list of locations, where each location has a suburb and postcode:

```
1  {
2      "locations": [
3          {
4              "suburb" : "Kensington",
5              "postcode" : 2033
6          },
7          {
8              "suburb" : "Mascot",
9              "postcode" : 2020
10         },
11         {
12             "suburb" : "Sydney CBD",
13             "postcode" : 2000
14         }
15     ]
16 }
```

Note:

- No trailing commas allowed
- Whitespace is ignored

# JSON - Writing & Reading

Python has powerful built in libraries to write and read json.

This involves converting JSON between a python-readable data structure, and a text-based dump of JSON

**json\_it.py**

**unjson\_it.py**

# YAML

*YAML* Ain't Markup Language (YAML) is a popular modern interchange format due it's ease of editing and concise nature. It's easy to convert between JSON and YAML online.

```
1 ---  
2 locations:  
3 - suburb: Kensington  
4   postcode: 2033  
5 - suburb: Mascot  
6   postcode: 2020  
7 - suburb: Sydney CBD  
8   postcode: 2000
```

Note:

- Like python, indentation matters
- A dash is used to begin a list item
- very common for configuration(s)

# XML

eXtensible Markup Language (XML) is more of a legacy interchange format being used less and less

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <root>
3   <locations>
4     <element>
5       <postcode>2033</postcode>
6       <suburb>Kensington</suburb>
7     </element>
8     <element>
9       <postcode>2020</postcode>
10      <suburb>Mascot</suburb>
11    </element>
12    <element>
13      <postcode>2000</postcode>
14      <suburb>Sydney CBD</suburb>
15    </element>
16  </locations>
17 </root>
```

# XML

Issues with XML include:

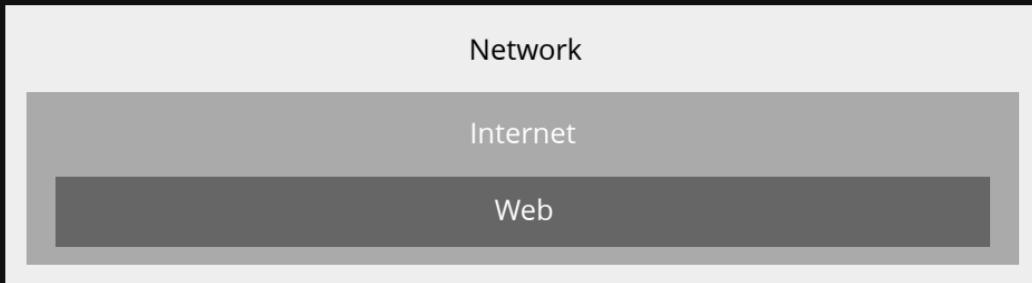
- More verbose (harder to read at a glance)
- More demanding to process/interpret
- More bytes required to store (due to open/closing tags)

While you will find very few modern applications choose to use XML as an interchange format, many legacy systems will still use XML as a means of storing data

# COMP1531

## 4.2 - Web - HTTP & Flask

# Computer Networks



# The network

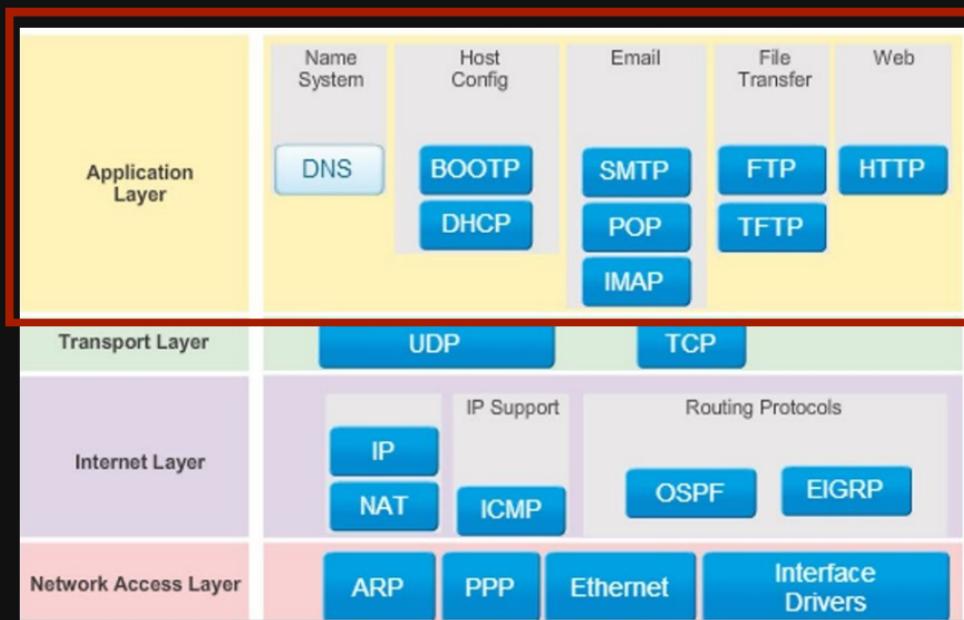
This is not a networking course:

- **Network:** A group of interconnected computers that can communicate
- **Internet:** A global infrastructure for networking computers around the entire world together
- **World Wide Web:** A system of documents and resources linked together, accessible via URLs

# Network Protocols

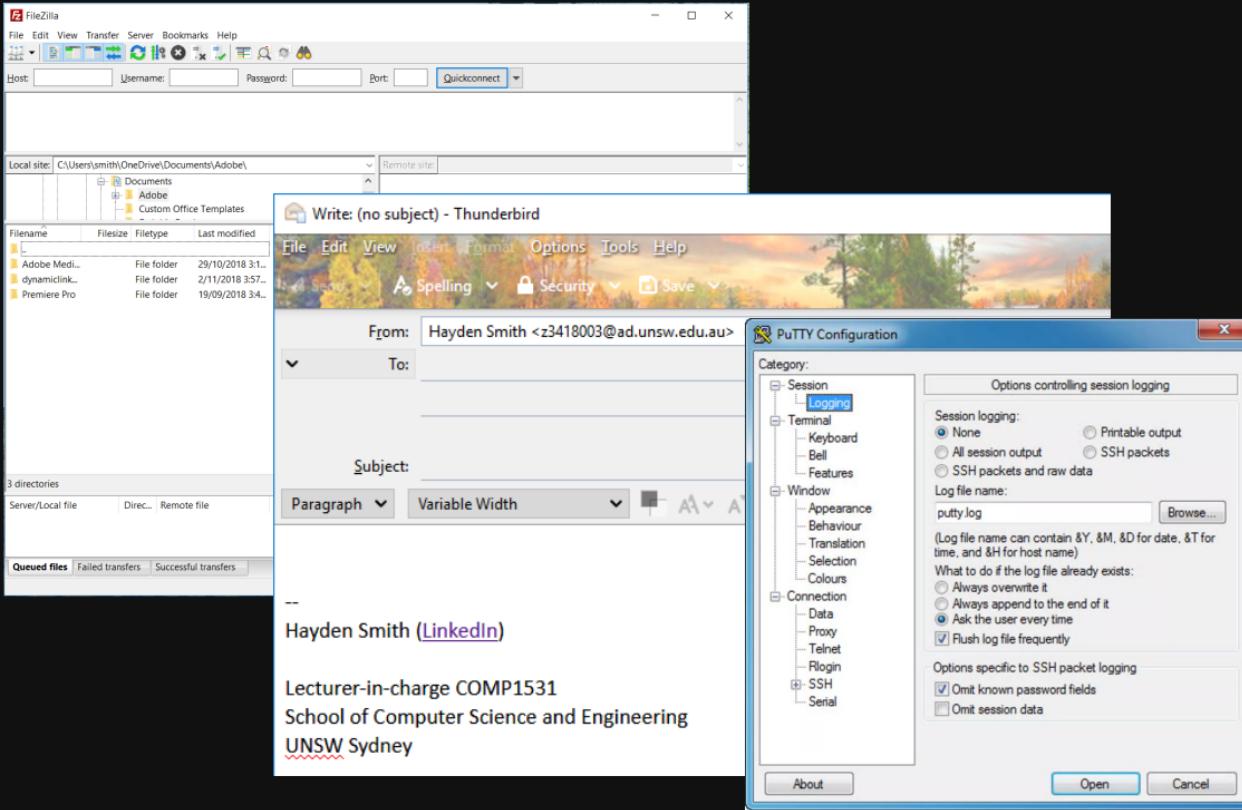
- Communication over networks must have a certain "structure" so everyone can understand
- Different "structures" (protocols) are used for different types of communication

# Network Protocols



Source

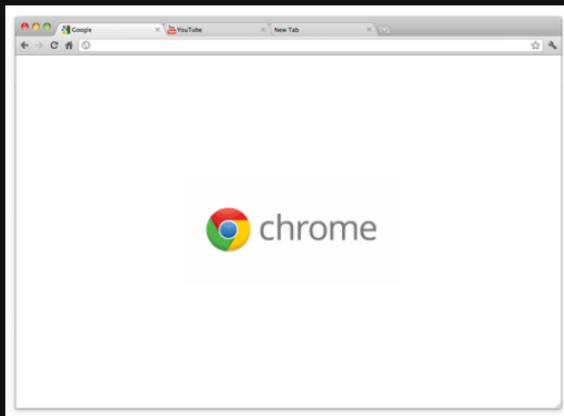
# Examples?



# HTTP

HTTP: Hypertext Transfer Protocol

I.E. Protocol for sending and receiving HTML documents (nowadays much more)



Request  
→

← Response



Web browsers are applications  
to request and receive HTTP

# HTTP Request & Response

## HTTP Request

```
1 GET /hello HTTP/1.1
2 Host: 127.0.0.1:5000
3 Connection: keep-alive
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36
6 Sec-Fetch-Mode: navigate
7 Sec-Fetch-User: ?1
8 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
9 Sec-Fetch-Site: none
10 Accept-Encoding: gzip, deflate, br
11 Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
```

## HTTP Response

```
1 HTTP/1.0 200 OK
2 Content-Type: text/html; charset=utf-8
3 Content-Length: 12
4 Server: Werkzeug/0.16.0 Python/3.5.3
5 Date: Wed, 09 Oct 2019 13:21:51 GMT
6
7 Hello world!
```

# Flask

Lightweight HTTP web server built in python

flask1.py

```
1 from flask import Flask
2 APP = Flask(__name__)
3
4 @APP.route("/")
5 def hello():
6     return "Hello World!"
7
8 if __name__ == "__main__":
9     APP.run()
```

```
1 $ python3 flask1.py
```

# Server an image

Time to serve an image via a flask server...

flask2.py

```
1 from flask import Flask, send_file
2 APP = Flask(__name__)
3
4 @APP.route("/img")
5 def img():
6     return send_file('./cat.jpg', mimetype='image/jpg')
7
8 if __name__ == "__main__":
9     APP.run()
```

```
1 $ python3 flask2.py
```

# Flask Reloading

Lightweight HTTP web server built in python

flask1.py

```
1 from flask import Flask
2 APP = Flask(__name__)
3
4 @APP.route("/")
5 def hello():
6     return "Hello World!"
7
8 if __name__ == "__main__":
9     APP.run()
```

Auto-refresh:

```
1 $ export FLASK_APP=flask1.py
2 $ export FLASK_DEBUG=1
3 $ export FLASK_RUN_PORT=0
4 $ python3 -m flask run
```

# Learn More

Some tutorials include:

1. <https://pythonspot.com/flask-web-app-with-python/>
2. <https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask>

When it comes to online tutorials, note that:

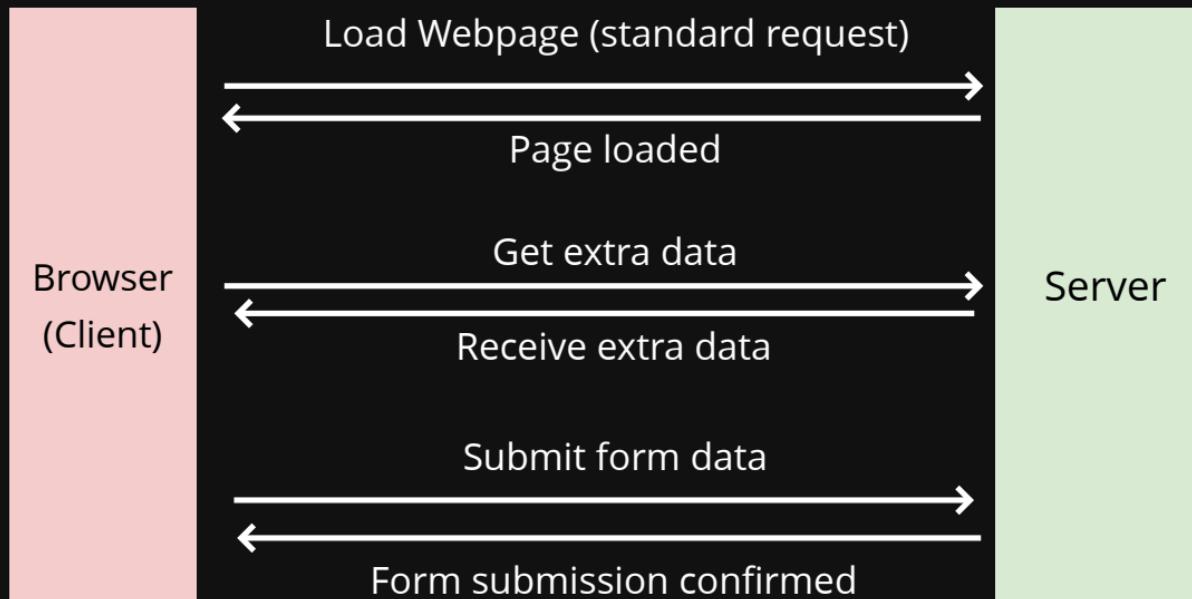
- Each "tutorial" may be using different python versions
- Each "tutorial" may have different aims in mind

# API

An API (Application Programming Interface) refers to an interface exposed by a particular piece of software.

The most common usage of "API" is for Web APIs, which refer to a "contract" that a particular service provides. The interface of the service acts as a black box and indicates that for particular endpoints, and given particular input, the client can expect to receive particular output.

# Web API



# Restful API & "CRUD"

A *RESTful API* is an application program interface (*API*) that uses HTTP requests to GET, PUT, POST and DELETE data. These 4 methods describe the "nature" of different API requests.

GET, PUT, POST, DELETE are HTTP Methods

Method	Operation
POST	Create
GET	Read
PUT	Update
DELETE	Delete

# Input & Output

Different CRUD properties require different approaches for input. All output are the same.

Inputs are either:

- GET: via URL and "request.args"
- PUT|POST|DELETE: via post-data and via "request.get\_json()"
- All outputs should be packaged up as JSON
- (JSON discussed later)

crud.py

```
1 from flask import Flask, request
2 from json import dumps
3
4 APP = Flask(__name__)
5
6 @APP.route("/one", methods=['GET'])
7 def one():
8     return dumps({
9         '1': request.args.get('data1'),
10        '2': request.args.get('data1'),
11    })
12
13 @APP.route("/two", methods=['POST'])
14 def two():
15     data = request.get_json()
16     return dumps({
17         '1': data['data1'],
18         '2': data['data2'],
19     })
20
21 if __name__ == '__main__':
22     APP.run()
```

# Using CRUD and state

## Task:

Create a web server that uses CRUD to allow you to  
create, update, read, and delete a point via HTTP  
requests

Use a global variable to manage the state.

**point.py**

# Talking to Flask

How can we talk to flask?

1. API client
2. Web Browser
3. URLLib via python

# API Client (ARC/Postman/Insomnia)

How to download/install postman:

- Open google chrome
- Google "ARC client"
- Install the addon and open it
- Follow the demo in the lectures

You may need to use a tool like this in the final exam.

# API Client (A R C)

ARC

## Request

HTTP request

Method: GET Request URL: An URL is required.

Socket

History

Send a request and recall it from here

Once you made a request it will appear in this place.

Saved

Save a request and recall it from here

Use **ctrl+s** to save a request. It will appear in this place.

Install new ARC with new features!

Selected environment: Default

SEND

Headers

Variables

Header name Header value

ADD HEADER

Headers are valid Headers size: bytes

# Web Browser

The screenshot shows a browser window with the URL `127.0.0.1:5000/hello`. The developer tools Network tab is open, displaying a timeline and detailed request information.

**Request Timeline:**

- 10 ms
- 20 ms
- 30 ms
- 40 ms
- 50 ms
- 60 ms
- 70 ms
- 80 ms
- 90 ms
- 100 ms
- 110 ms

**Request Details:**

- Name:** hello
- General:**
  - Request URL: `http://127.0.0.1:5000/hello`
  - Request Method: GET
  - Status Code: 200 OK
  - Remote Address: 127.0.0.1:5000
  - Referrer Policy: no-referrer-when-downgrade
- Response Headers:**
  - Content-Length: 12
  - Content-Type: text/html; charset=utf-8
  - Date: Wed, 09 Oct 2019 13:26:05 GMT
  - Server: Werkzeug/0.16.0 Python/3.5.3
- Request Headers:**
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3
  - Accept-Encoding: gzip, deflate, br
  - Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
  - Cache-Control: max-age=0

1 requests | 166 B transferred

# **requests** - Python

**requests** is a python3 library that allows you to programmatically make HTTP requests to a web server.

You will use this extensively in iteration  
2.

# requests - Python

## echo.py

```
1 from flask import Flask, request
2 from json import dumps
3
4 APP = Flask(__name__)
5
6 @APP.route("/echo", methods=['GET'])
7 def echo():
8     return dumps({'data': request.args.get('data')})
9
10 if __name__ == '__main__':
11     APP.run()
```

## echo\_main.py

```
1 import json
2 import requests
3
4 def get_payload():
5     response = requests.get('http://127.0.0.1:5000/echo', params={'data': 'hi'})
6     payload = response.json()
7     print(payload)
8
9 if __name__ == '__main__':
10     get_payload()
```

We expect you to do your own research for POST

# Web server as a wrapper

Because you've written so many **integration** tests  
for iteration 1, it makes sense to:

1. Implement all of the functions from iteration one
2. *Then* wrap them in a flask server

# Web server as a wrapper

## iter2example/search.py

```
1 def search(token, query_str):
2     return {
3         'messages' : [
4             'Hello ' + token + ' ' + query_str,
5             # Not the right structure
6         ]
7     }
```

## iter2example/server.py

```
1 from json import dumps
2 from flask import Flask, request
3
4 from search import search_fn
5
6 APP = Flask(__name__)
7
8 @APP.route('/search', methods=['GET'])
9 def search_flask():
10     return dumps(search(requests.args.get('token'), request.args.get('query_str')))
11
12 if __name__ == '__main__':
13     APP.run()
```

# (Bonus) interesting question

How do companies track whether or  
not you've read an email they've sent  
you?

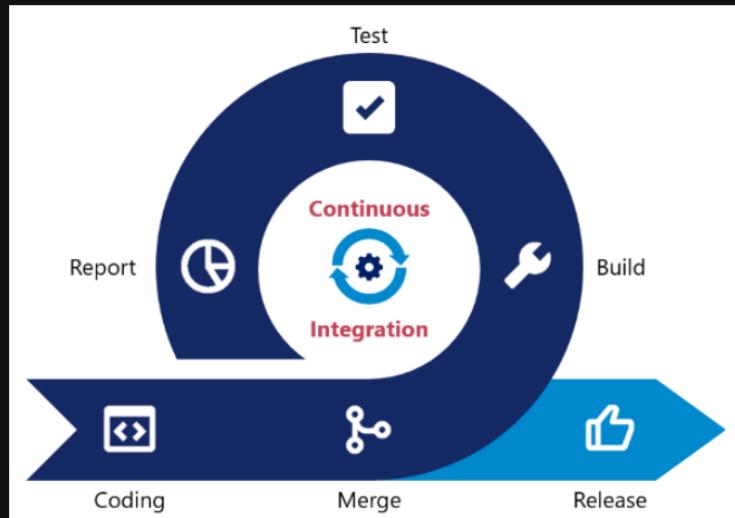
# COMP1531

## 5.1 - SDLC Testing - Continuous Integration

# Continuous Integration

**Continuous integration:** Practice of automating the integration of code changes from multiple contributors into a single software project.

# Continuous Integration



# Continuous Integration

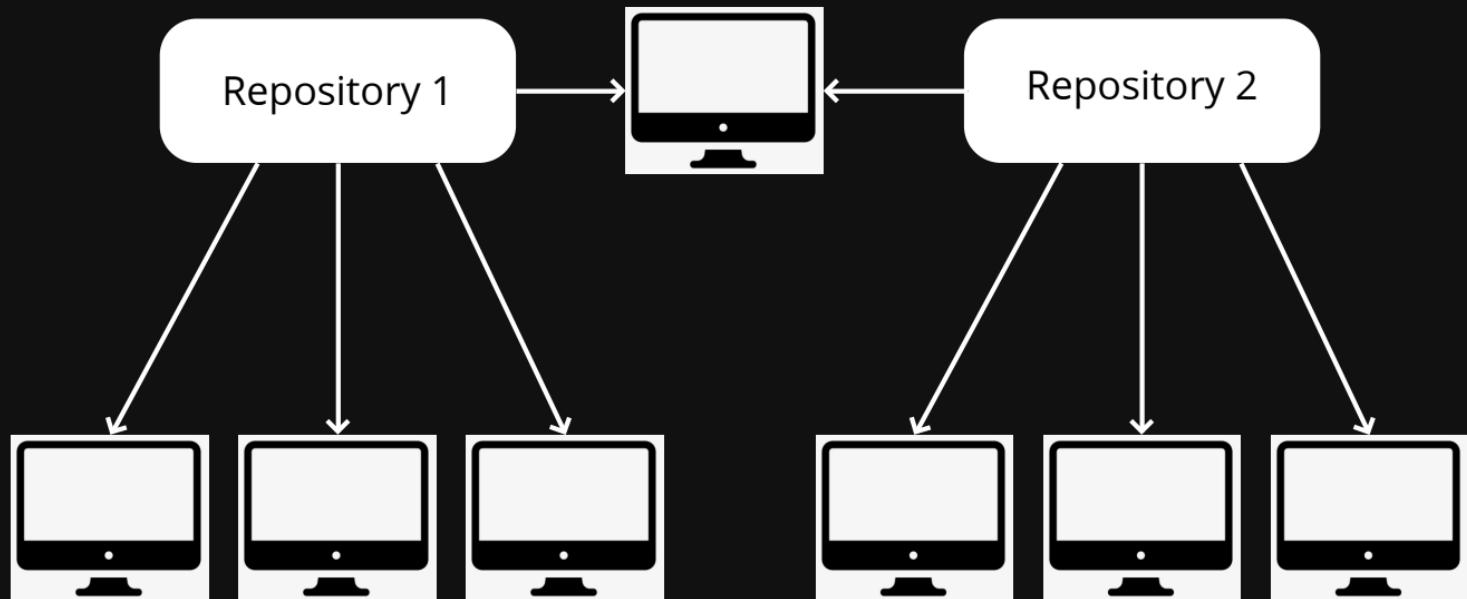
## **Key principles and processes:**

1. Write tests:
  1. Ideally tests for each story
  2. Broad tests: unit, integration, acceptance, UI tests
2. Use code coverage checkers
3. Merge and integrate code as often as possible
4. Ensure the build always works (i.e. is "green")

# How it works

- Typically tests will be run by a "runner", which is an application that works with your version control software (git) to execute the tests. This is because tests can require quite resource intensive activities
  - Gitlab: No runners built in
  - Bitbucket: Runners built in

# Broad Architecture



# CI: Readings

We will assume you have read the following items:

- <https://about.gitlab.com/product/continuous-integration/>
- <https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration>

# Continuous integration, gitlab

Gitlab, like many source control tools, has a way of doing continuous integration. An [overview is here](#) and a [start guide is here](#).

There is quite a lot of variance and depth to this, so we will not cover it in any detail besides high level

A simple example can be found [here](#).

# Continuous integration, gitlab

Let's do a demo setting up CI with gitlab.

# COMP1531

## 5.2 - HTTP - Auth & Auth

# Auth vs Auth

**Authentication:** Process of verifying the identity of a user

**Authorisation:** Process of verifying an identity's access privileges

# Authentication

Naive method:

- User registers, we store their password
- When user logs in, we compare their input password to their stored password

Let's observe *auth.py*  
(found in lectures repo)

# Authentication

What's wrong with this?

# Authentication

Using **hashlib** to create a hash

hash.py

```
1 import hashlib
2 print("mypassword")
3 print("mypassword".encode())
4 print(hashlib.sha256("mypassword".encode()))
5 print(hashlib.sha256("mypassword".encode()).hexdigest())
```

# Authentication

**Now let's improve auth.py**

# Authorisation

Authorisation typically involves giving the user some kind of pseudo-password that they store on their computer (client-side) which is a shortcut method for authorising a particular user.

An SSH key is an example of this.

# Authorisation

## What is a "token"?

A packet of data used to authorise the user.

## What kind of tokens exist?

- **User ID:** The ID number of the particular user.
- **JWT'd User ID:** The ID number of a particular user stored in a JWT.
- **Session:** Some kind of ID representing that unique login event, whereby the session is tied to a user ID.
- **JWT's Session:** Some kind of ID representing a session that is stored in a JWT.

# Authorisation

	<b>User ID</b>	<b>Session ID</b>
Non JWT	One login session + insecure	Concurrent login sessions + insecure
JWT	One login session + secure	Concurrent login sessions + secure

# What is a JWT?

*"JSON Web Tokens are an open, industry standard [RFC 7519](#) method for representing claims securely between two parties."*

They are lightweight ways of encoding and decoding  
private information via a secret

Play around:  
<https://jwt.io/>

# Let's practice with python

Using a JWT in python:

<https://pyjwt.readthedocs.io/en/latest/>

webtoken.py

```
1 import jwt
2
3 SECRET = 'sempai'
4
5 encoded_jwt = jwt.encode({'some': 'payload'}, SECRET, algorithm='HS256').decode('utf-8')
6 print(jwt.decode(encoded_jwt.encode('utf-8'), SECRET, algorithms=[ 'HS256']))
```

# Let's practice with python

Now let's improve auth.py

# COMP1531

## 5.3 - SDLC Design - Software Engineering Principles

# Design Smells

- **Rigidity:** Tendency to be too difficult to change
- **Fragility:** Tendency for software to break when single change is made
- **Immobility:** Previous work is hard to reuse or move
- **Viscosity:** Changes feel very slow to implement
- **Opacity:** Difficult to understand
- **Needless complexity:** Things done more complex than they should be
- **Needless repetition:** Lack of unified structures
- **Coupling:** Interdependence between components

# Design Principles

Purpose is to make items:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

Often, this is achieved through **abstraction**. Abstraction is the process of removing characteristics of something to reduce it some a more high level concept

e.g. Using variable, "counter" rather than "1", ...

# DRY

"Don't repeat yourself" (DRY) is about reducing repetition in code. The same code/configuration should ideally not be written in multiple places.

Defined as:

*"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"*

# DRY

How can we clean this up?

```
1 import sys
2
3 if len(sys.argv) != 2:
4     sys.exit(1)
5
6 num = int(sys.argv[1])
7
8 if num == 2:
9     for i in range(10, 20):
10         result = i ** 2
11         print(f"{i} ** 2 = {result}")
12
13 elif num == 3:
14     for i in range(10, 20):
15         result = i ** 3
16         print(f"{i} ** 3 = {result}")
17
18 else:
19     sys.exit(1)
```

# DRY

How can we improve this?

```
1 import jwt
2
3 encoded_jwt = jwt.encode({'some': 'payload'}, 'applepineappleorange', algorithm='HS256')
4 print(jwt.decode(encoded_jwt, 'applepineappleorange', algorithms=['HS256']))
```

# KISS

"Keep it Simple, Stupid" (KISS) principles state that a software system works best when things are kept simple. It is the belief that complexity and errors are correlated.

Your aim should often be to use the simplest tools to solve a problem in the simplest way.

# KISS

**Example 1:** Write a python function to generate a random string with up to 50 characters that consist of lowercase and uppercase characters

# KISS

**Example 2:** Write a function that prints what day of the week it is today

<https://stackoverflow.com/questions/9847213/how-do-i-get-the-day-of-week-given-a-date-in-python>

# KISS

## **Example 3:** Handling command line arguments

```
1 python3 commit.py -m "Message"  
2 python3 commit.py -am "All messages"
```



# Encapsulation

*Similar to  
Abstraction,  
but hide interface,  
like ADT.*

**Encapsulation:** Maintaining type abstraction by restricting direct access to internal representation of types (types include classes)

# Encapsulation

## Example:

Consider this code:

```
1 class Point:  
2     def __init__(self, x,y):  
3         self.x = x  
4         self.y = y  
5  
6 def distance(start, end):  
7     return sqrt((end.x - start.x)**2 + (end.y - start.y)**2)
```

What if we wanted to store points in polar coordinates?

# Encapsulation

## Example:

Consider this code:

```
1 class Queue:
2     def __init__(self):
3         self.entries = []
4
5     def enqueue(self, entry):
6         self.entries.append(entry)
7
8     def dequeue(self):
9         return self.entries.pop(0)
```

Can we prevent stealing spots in the queue?

# Top-down thinking

Similar to "You aren't gonna need it" (YAGNI) that says a programmer should not add functionality until it is needed.

Top-down thinking says that when building capabilities, we should work from high levels of abstraction down to lower levels of abstraction.

Implement front-end first,

Based on the requirement,  
↓  
build back-end.

Rather than inverted way.

- X Not building a lot of functions,
- X then choose what front-end needs.

# Top-down thinking

**Question 1:** Given two Latitude/Longitude coordinates, find out what time I would arrive at my destination if I left now. Assume I travel at the local country's highway speed

# Why is well designed software important?



- When you only do this loop once, writing bad code has minimal impacts
- When we complete this "cycle" many times, modifying bad code comes at a high cost

# Why is well designed software important?

*"Poor software quality costs more than \$500 billion per year worldwide" – Casper Jones*

*Systems Sciences Institute at IBM found that it costs **four- to five-times as much** to fix a software bug after release, rather than during the design process*

# Why do we write bad code?

Often, our default tendency is to write bad code. Why?

- It's quicker not to think too much about things
  - Good code requires thinking not just about now, but also the future
- Pressure from business we're looking for
- Refactoring takes time

**Bad code:** Easy short term, hard long term

**Good code:** Hard short term, easy long term

# Why do we want to write good code?

- More consistent with Agile Manifesto
  - "Welcome changing requirements"
- Adapt easier to the natural SD life cycle

# Refactoring

Restructuring existing code *without* changing its  
external behaviour.

Typically this is to fix code or design smells and thus  
make code more *Maintainable*

# Finding a balance

- Don't over-optimize to remove design smells
- Don't apply principles when there are no design smells - unconditional conforming to a principle is a bad idea, and can sometimes add complexity back in

# COMP1531

## 5.4 - SDLC Development - Persistence

# Data

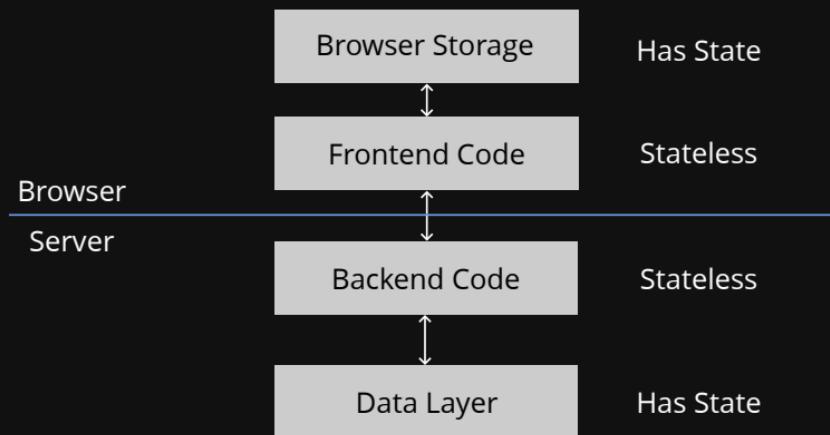
**Data:** Facts that can be recorded and have implicit meaning

Data is one of the fastest and most rapidly growing areas within software.

From **data (raw)** we can find **insights (information)** that allow us to make **decisions**.

# Data Layer

**Data Layer:** The part of the tech stack that provides persistence



# Databases

Data is only as powerful as you can store and access it. Study COMP3311 to learn more about efficient data storage.

There are 3 main ways to store data:

1. In-memory (non-persistent)
2. In-file
3. In-database (SQL)

As you move down the list, barrier to entry becomes higher, but so does performance.

In COMP1531 we will only explore (2)

# Storing Data: Persistence

**Persistence:** When program state outlives the process that created it. This is achieved by storing the state as data in computer data storage

## What is storage?

- CPU cache?
- RAM?
- Hard disk? (we usually mean this one)

# Storing Data: Persistence

Most modern backend/server  
applications are just source code + data

# Storing Data: In practice

A very common and popular method of storing data in python is to "pickle" the file.

- Pickling a file is a lot like creating a .zip file for a variable.
- This "variable" often consists of many nested data structures (a lot like your iteration 2 data)

# Storing Data: In practice

Let's look at an example

**pickle\_it.py**

**unpickle\_it.py**

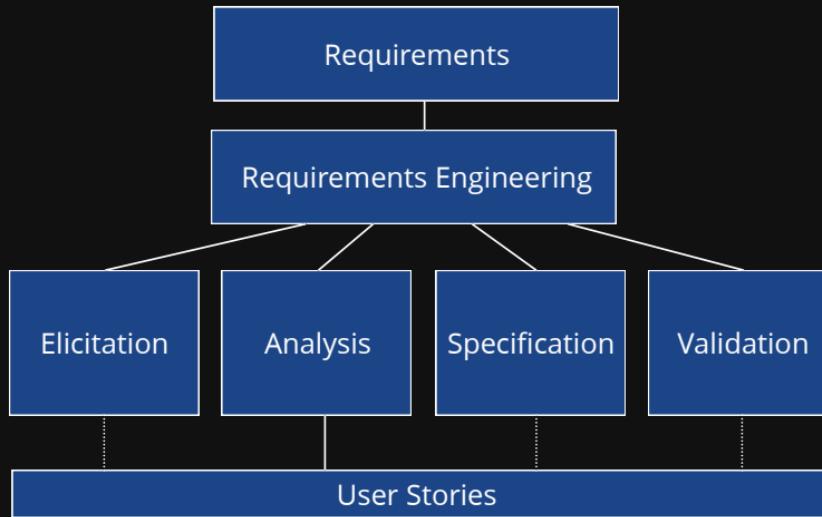
# COMP1531

## 7.1 - SDLC Requirements - Overview

# SDLC



# Requirements



# Requirements

IEEE defines a requirement as:

**A condition or capability needed by a user to solve  
a problem or achieve an objective**

We would also describe requirements as:

- Agreement of work to be completed by all stakeholders
- Descriptions and constraints of a proposed system

# Functional v Non-Functional

**Functional requirements** specify a specific capability/service that the system should provide. It's what the system does.

**Non-functional requirements** place a constraint on how the system can achieve that. Typically this is a performance characteristic.

Great reading on the topic

# Functional v Non-Functional

**For example:**

Functional: The system must send a notification to all users whenever there is a new post, or someone comments on an existing post

本职工作.

Non-functional: The system must send emails no later than 30 minutes after from such an activity

Restriction .

# Requirements Engineering

We need a durable process to determine requirements

*"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting systems if done wrong"*  
*(Brooks, 1987)*

# Requirements Engineering

Requirements Engineering is:

- A **set of activities** focused on identifying the purpose and goal of a software system
- A **negotiation process** where stakeholders agree on what they want. Stakeholders include:
  - End user(s)
  - Client(s) (often businesses)
  - Design team(s)

# Requirements Engineering

Requirements engineering often follows a logical process across 4 steps:

提纲

1. Elicitation of raw requirements from stakeholders
2. Analysis of requirements
3. Formal specification of requirements
4. Validation of requirements

# RE | Step 1 | Elicitation

## **Questions and discovery**

- Market Research
- Interviews with Stakeholders
- Focus groups
- Asking questions "What if? What is?"

# RE | Step 2 | Analysis

## **Building the picture**

- Identify dependencies, conflicts, risks
- Establish relative priorities
- Usually done through:
  - User stories (discussed today)
  - Use cases (discussed next week)

# RE | Step 3 | Specification

## **Refining the picture**

- Establishing the right sense of granularity
  - There is no perfect way to granulate
- Often the stage of breaking up into functional and non-functional
- E.G. Try and granulate "The system shall keep the door locked at all times, unless instructed otherwise by an authorised user. When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed)"

# RE | Step 4 | Validation

Going back to stakeholders and  
ensuring requirements are correct

# Challenges during RE?

What are some challenges we may face while engaging in Requirements engineering?

- Requirements sometimes only understood after design/build has begun
- Clients/customers sometimes don't know what they want
- Clients/customers sometimes change their mind
- Developers might not understand the subject domain
- Limited access to stakeholders
- Jumping into details or solutions too early (XY problem)

# Let's step through an example



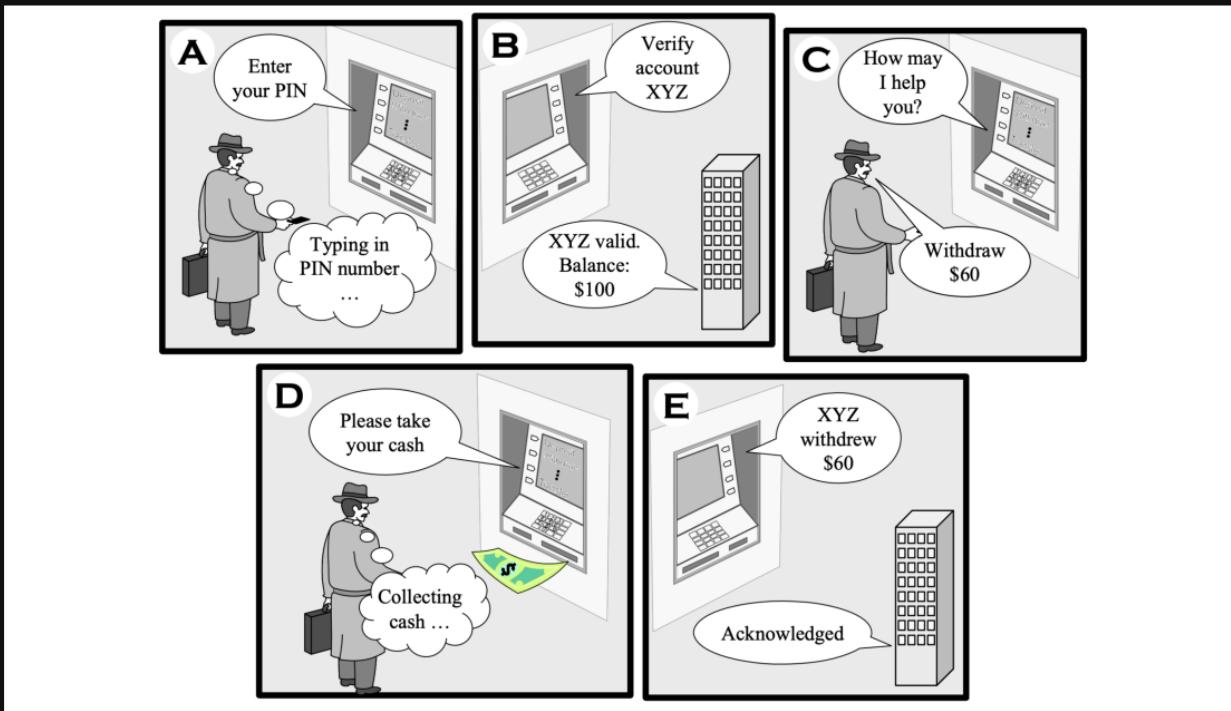
# COMP1531

## 7.2 - SDLC Requirements - Use Cases

# Use cases

- Represent a *dialogue* between the user and the system, with the aim of helping the user achieve a business goal
- The user initiates *actions* and the system responds with *reactions*
- They consider the system a black box

# Example



# Representations

- Informal list of steps
- Use-Case diagrams
- Cockburn style (not required reading)

# Initial template

- **Use Case:** <the name should be the goal as a short active verb phrase>
- **Goal in Context:** <a longer statement of the goal, if needed>
- **Scope:** <what system is being considered black-box under design>
- **Level:** <one of: Summary, Primary task, Subfunction>
- **Preconditions:** <what we expect is already the state of the world>
- **Success End Condition:** <the state of the world upon successful completion>
- **Failed End Condition:** <the state of the world if goal abandoned>
- **Primary Actor:** <a role name for the primary actor, or description>
- **Trigger:** <the action upon the system that starts the use case, may be time event>

# ATM Example

- **Use Case:** Withdraw Money
- **Goal in Context:** Customers need to withdraw money from their accounts without entering the bank
- **Scope:** ATM, banking infrastructure
- **Level:** Primary Task
- **Preconditions:** The customer has an account with the bank
- **Success End Condition:** The customer has the money they needed to withdraw
- **Failed End Condition:** The customer has no money
- **Primary Actor:** Customer
- **Trigger:** Customer puts card into ATM

# Steps taken

## MAIN SUCCESS SCENARIO

<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>

<step #> <action description>

# ATM Example

## MAIN SUCCESS SCENARIO

Step 1. ATM asks customer for pin

Step 2. Customer enters pin

Step 3. ATM asks bank to verify pin and account

Step 4. Bank informs ATM of validity and balance of account

Step 5. ATM asks customer what action they wish to take

Step 6. Customer asks to withdraw an amount of money

Step 7. ATM Dispenses money to customer

Step 8. ATM informs bank of withdrawal

# In More Depth

- Can be used to model variations in steps (e.g. Insufficient funds)
- If you wish to know more about use cases, see here:
  - Software Engineering - Ivan Marsic (Chapter 2, Section 4)
  - <http://www.cs.otago.ac.nz/coursework/cosc461/uctempla.htm>
  - Writing Effective Use Cases - Alistair Cockburn

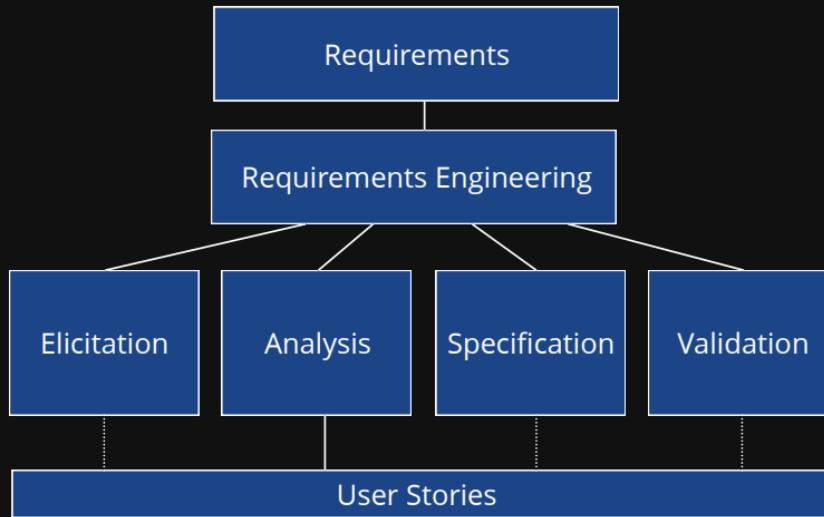
# **COMP1531**

## **7.3 - SDLC Requirements - User Stories & UAT**

# SDLC



# Requirements



# User Stories - Overview

**User Stories are a method of requirements engineering used to inform the development process and what features to build with the user at the centre.**

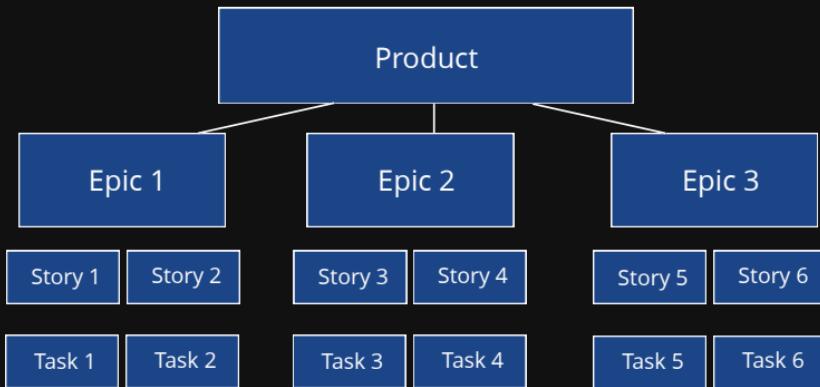
# User Stories - Structure

When a customer tells you what they want, try and express it in the form **As a < type of user >, I want < some goal > so that < some reason >**

**E.G. They say:**

- E.G. They say:
  - A student can purchase monthly parking passes online
- But your story becomes:
  - As a **student**, I want to **purchase a parking pass** so that **I can drive to school**

# User Stories - Structure



# User Stories - Nature

## **User stories:**

- Are written in non-technical language
- Are user-goal focused, not product-feature focused
  - User stories inform feature decisions

## **Why do we care?**

- They keep customers at the centre
- Keep it problem focused, not solution focused

# User Stories - Activity

**Building a to-do list**

# User Stories - More

**Read more about user stories here:**

<https://www.atlassian.com/agile/project-management/user-stories>

**How do we know  
we've met the user  
story requirement?**

# INVEST

- I = Independent: user story could be developed independently and delivered separately
- N = Negotiable: avoid too much detail.
- V = Valuable: must hold some value to the client
- E = Estimable: *we'll get to this in a later lecture*
- S = Small: user story should be small
- T = **Testable**

# User Acceptance Criteria

- Break down a user story into criteria that must be met for the user, or customer, to accept
- Written in natural language
- Can be refined before implementation

# Example

*As a user, I want to use a search field to type a city, name, or street, so that I can find matching hotel options.*

- The search field is placed on the top bar
- Search starts once the user clicks “Search”
- The field contains a placeholder with a grey-colored text: “Where are you going?”
- The placeholder disappears once the user starts typing
- Search is performed if a user types in a city, hotel name, street, or all combined
- The user can't type more than 200 symbols

# Best practices

- Acceptance criteria should not be too broad
- ... but nor should they be too narrow
- Minimise technical detail
  - They can be more technical than the story itself, but client still needs to understand them
- While they can be updated during development, they should first be written *before* it starts

# From Criteria to Testing

- *Acceptance Tests* are tests that are performed to ensure acceptance criteria have been met
- Not all acceptance criteria can easily be mapped to *automated* acceptance tests
- Acceptance tests are *black-box* tests

## Example 2:

*As a user, I can log in through a social media account, because I always forget my passwords*

- Can log in through Facebook
- Can log in through LinkedIn
- Can log in through Twitter

# Scenario Oriented AC

- The Acceptance criteria from before are often referred to a rule-based AC
- Sometimes it is preferable to have AC that describe a scenario
- This can be done in the Given/When/Then format:
  - *Given* some precondition
  - *When* I do some action
  - *Then* I expect some result

# Example 3:

*As a user, I want to be able to recover the password to my account, so that I will be able to access my account in case I forgot the password.*

**Scenario:** Forgot password

**Given:** The user has navigated to the login page

**When:** The user selected forgot password option

**And:** Entered a valid email to receive a link for password recovery

**Then:** The system sent the link to the entered email

**Given:** The user received the link via the email

**When:** The user navigated through the link received in the email

**Then:** The system enables the user to set a new password

# Which one to use?

- Rule-based acceptance criteria are simpler and generally work for all sorts of stories
- Scenario-based AC work for stories that imply specific user actions, but don't work for higher-level system properties (e.g. design)
- Scenario-based AC are more likely to be implementable as tests

# Further reading

- <https://www.mountaingoatsoftware.com/blog/the-two-ways-to-add-detail-to-user-stories>
- <https://www.altexsoft.com/blog/business/acceptance-criteria-purposes-formats-and-best-practices/>
- <https://dzone.com/articles/acceptance-criteria-in-software-explanation-exampl>

# COMP1531

## 7.4 - SDLC Design - System Modelling

What's a model?

# Conceptual Modelling

- *A model that is conceptual*
  - ... *with a real world correspondence*
  - ... *without a real world correspondence*
- *A model of a concept*

# Conceptual models software engineers care about

- Data models
- Mathematical models
- Domain models
- Data flow models
- State transition models (today)

# How models are used

- To predict future states of affairs.
- Understand the current state of affairs.
- Determine the past state of affairs.
- **To convey the fundamental principles and basic functionality of systems (communication)**

# Communicating models

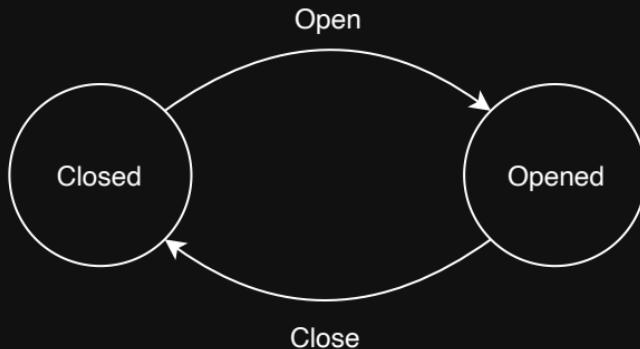
- Four fundamental objectives of communicating with a conceptual model:
  1. Enhance an individual's understanding of the representative system
  2. Facilitate efficient conveyance of system details between stakeholders
  3. Provide a point of reference for system designers to extract system specifications
  4. Document the system for future reference and provide a means for collaboration

# System Modelling

- Structural – Emphasise the static structure of the system
  - UML class diagrams
  - ER diagrams
  - ... many others
- Behavioural - Emphasise the dynamic behaviour
  - State diagrams
  - Use case diagram
  - ... some others

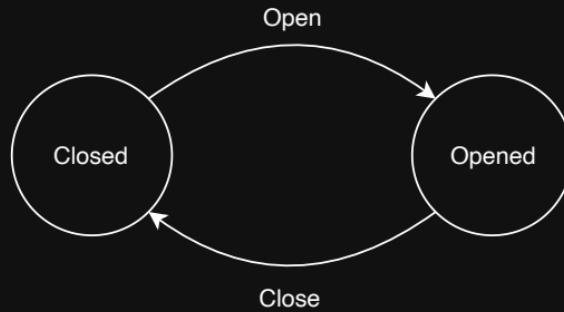
# State Machines

- Machines made up of a finite number of states.
- The machine can be *transitioned* from one state to another
- Simple example: a door



# State diagrams

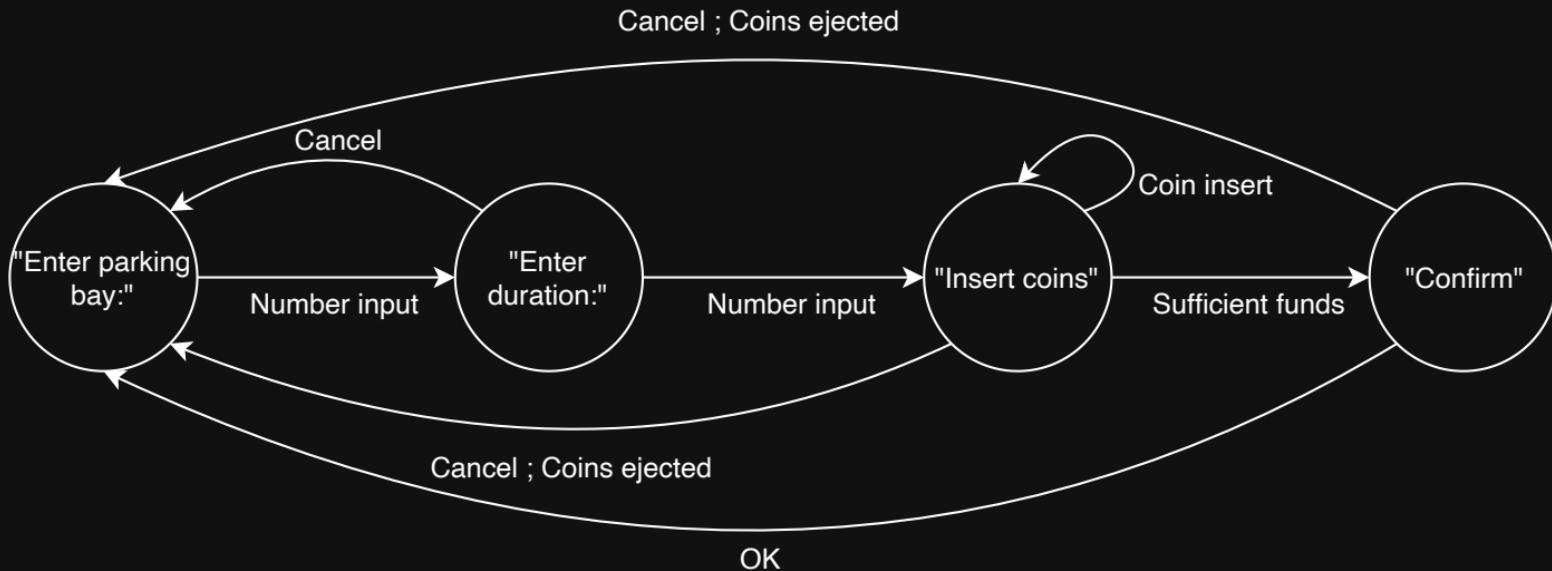
- A diagrammatic representation of a state.
- Some variation in notation.
- Typically: states are circles, transitions are labelled arrows connecting them



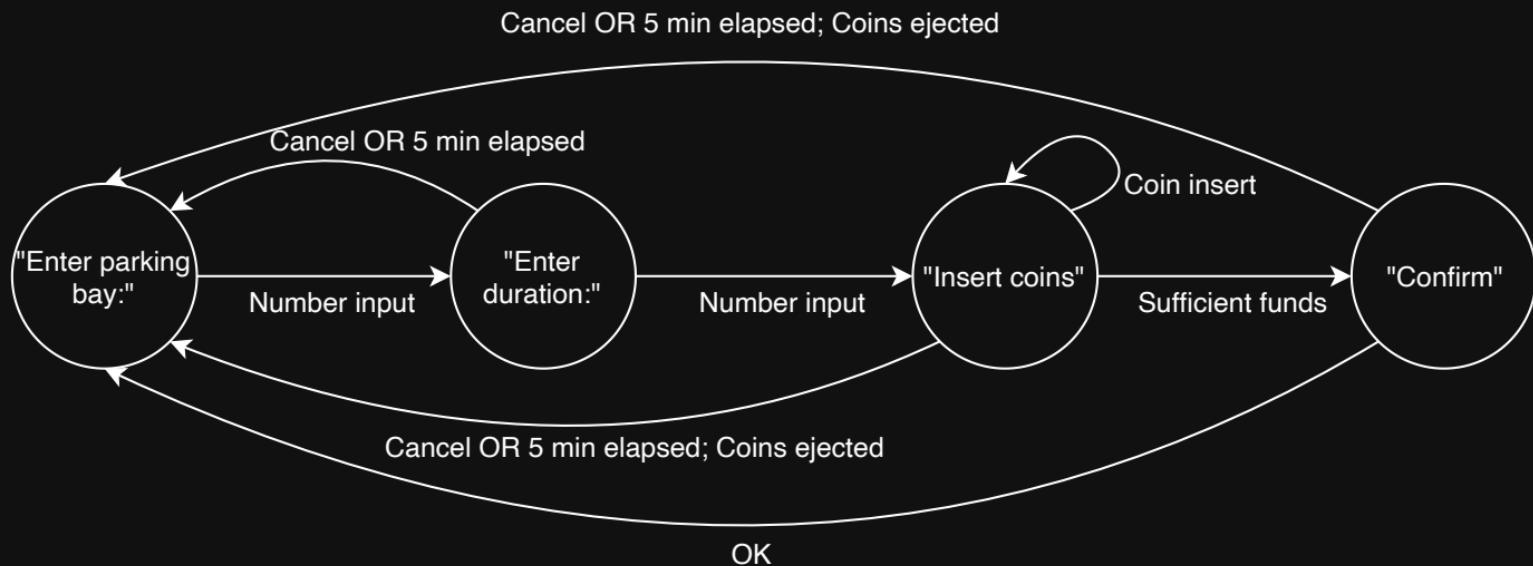
# State machines

- Useful for modelling systems that have clearly defined states. For example:
  - UIs with different screens
  - Network protocols
  - Conversational interfaces

# Parking meter



# Parking meter



# Opal Card

- Can we model the opal card system as a state machine?

# COMP1531

## 7.5 - GIT - Resets & Amending

# Mistakes

- Everything we've done until this point continues to *build* on the git history. But we've largely considered the git history immutable.
- With git, sometimes we make mistakes. Sometimes we want to undo things, or **change history**.

Two ways we're going to discuss this are:

- git **resets**
- git commit **amend**

# git reset

- Atlassian has a very clear article about git reset. We will use this as guidance.
- We will mainly discuss **hard** and **soft** resets.
- Let's do a demonstration.

# git commit --amend

```
git commit --amend -m "Commit"
```

- Sometimes we need to update our previous commit name. We can do that easily by making another commit that overrides it.
- Let's do a demonstration.

# COMP1531

## 7.6 - SDLC Design - SE Principles 2

# Decorators

Decorators allow us to add functionality to a function without altering the function itself, by "decorating" (wrapping) around it.

But first... some background

# Function Parameters

decor1.py

```
1 def fool(zid, name, age, suburb):
2     print(zid, name, age, suburb)
3
4 def foo2(zid=None, name=None, age=None, suburb=None):
5     print(zid, name, age, suburb)
6
7 if __name__ == '__main__':
8
9     fool('z3418003', 'Hayden', '72', 'Kensington')
10
11    foo2('z3418003', 'Hayden')
12    foo2(name='Hayden', suburb='Kensington', age='72', zid='z3418003')
13    foo2(age='72', zid='z3418003')
14
15    foo2('z3418003', suburb='Kensington')
```

# Function Parameters

decor2.py

```
1 def foo(zid=None, name=None, *args, **kwargs):
2     print(zid, name)
3     print(args) # A list
4     print(kwargs) # A dictionary
5
6 if __name__ == '__main__':
7
8     foo('z3418003', None, 'mercury', 'venus', planet1='earth', planet2='mars')
```

decor3.py

```
1 def foo(*args, **kwargs):
2     print(args) # A list
3     print(kwargs) # A dictionary
4
5 if __name__ == '__main__':
6     foo('this', 'is', truly='dynamic')
```

# A proper decorator

decor4.py

```
1 def make_uppercase(input):
2     return input.upper()
3
4 def get_first_name():
5     return "Hayden"
6
7 def get_last_name():
8     return "Smith"
9
10 if __name__ == '__main__':
11     print(make_uppercase(get_first_name()))
12     print(make_uppercase(get_last_name()))
```

# A proper decorator

decor5.py

This code can be used as a template

```
1 def make_uppercase(function):
2     def wrapper(*args, **kwargs):
3         return function(*args, **kwargs).upper()
4     return wrapper
5
6 @make_uppercase
7 def get_first_name():
8     return "Hayden"
9
10 @make_uppercase
11 def get_last_name():
12     return "Smith"
13
14 if __name__ == '__main__':
15     print(get_first_name())
16     print(get_last_name())
```

# Decorator, run twice

decor6.py

```
1 def run_twice(function):
2     def wrapper(*args, **kwargs):
3         return function(*args, **kwargs) \
4             + function(*args, **kwargs)
5     return wrapper
6
7 @run_twice
8 def get_first_name():
9     return "Hayden"
10
11 @run_twice
12 def get_last_name():
13     return "Smith"
14
15 if __name__ == '__main__':
16     print(get_first_name())
17     print(get_last_name())
```

# Decorator, more

## decor7.py

```
1 class Message:
2     def __init__(self, id, text):
3         self.id = id
4         self.text = text
5
6 messages = [
7     Message(1, "Hello"),
8     Message(2, "How are you?"),
9 ]
10
11 def get_message_by_id(id):
12     return [m for m in messages if m.id == id][0]
13
14 def message_id_to_obj(function):
15     def wrapper(*args, **kwargs):
16         argsList = list(args)
17         argsList[0] = get_message_by_id(argsList[0])
18         args = tuple(argsList)
19         return function(*args, **kwargs)
20     return wrapper
21
22 @message_id_to_obj
23 def printMessage(message):
24     print(message.text)
25
26 if __name__ == '__main__':
27     printMessage(1)
```

# Single Responsibility Principle

Every module/function/class in a program should have **responsibility** for just a **single** piece of that program's functionality

# Single Responsibility Principle

## Functions

We want to ensure that each function is only responsible for one task.  
If it's not, break it up into multiple functions.

This is often a good idea. The only instances where this might not be a good idea are if it complicates the **caller** substantially (i.e. makes the code calling your split up functions overly complex)

*Primary purpose: Readability and modularity*

# Single Responsibility Principle

## Classes

Three files:

- srp2.py: Poor SRP
- srp2\_fixed.py: Fixed SRP, abstraction remains
- srp2\_fixed2.py: Fixed SRP, no abstraction

We can apply the same principles to classes, ensuring that a single class maintains a single broad responsibility, and each function within the class also has a more specific single responsibility

# COMP1531

## 8.1 - Web - (Bonus) Frontend Basics

# Front-end Design

Building front-ends are not covered in COMP1531. CSE offers a few courses relating to them though:

- UX & Design: COMP3511, COMP4511
- Technical: COMP6080

These skills will help prevent you building **bad** interfaces.

# Web Applications

- At least 2 major components (client, server)
- Multiple languages involved
  - Javascript, HTML, CSS, and whatever language implements your server.
- Fundamentally different to the sorts of programs you may have written so far

Today we're going to write a very basic web application.

# Web Browser

- HTML, CSS, Javascript all are languages with internationally determined standards
  - How are decisions made on these standards?
  - Who pays for all these software engineers to do the work?
- Web browsers implement these standards and allow us to run HTML/CSS/Javascript

# HTML

- What does HTML stand for?
- HTML is the standard markup language for webpages
- HTML5 brought with it a massive wave of changes
- Let's make the following with HTML:
  - a basic page
  - headers, lists

# CSS

- CSS focuses on **styling**
- It provides styles in a format of:
  - attribute: value
- Styles can either be:
  - Part of the HTML in the style tag
  - Inline in the HTML file
  - External in a .css file
- Let's restyle our previous page using div/spans and CSS

# Javascript

Javascript is the "programming language" of front-ends. The interpreters for the language are built directly into your web browser. Most modern web browsers build their Javascript interpreter on top of the popular V8 engine.

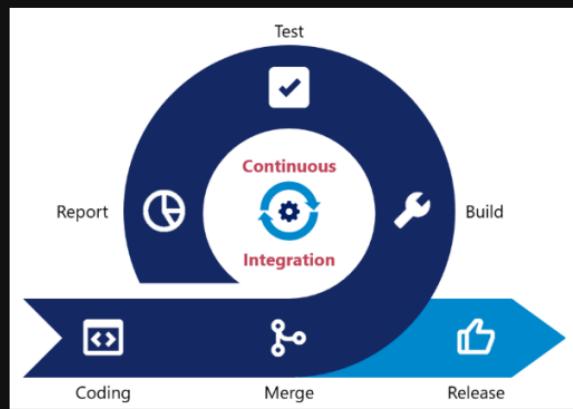
We can use Javascript to make API calls and display results.

# COMP1531

## 8.2 - SDLC Deployment

# Continuous Integration

**Continuous integration:** Practice of automating the integration of code changes from multiple contributors into a single software project.



# Software Deployment

Deployment: Activities relating to making a **software system available for use.**



Diagrams sourced from atlassian, gitlab, microsoft

# Simple example: CSE

Every CSE student has a **public\_html** folder that is exposed to the internet.

# Historical Deployment

Historically, **deployment** was a much less frequently occurring process.

Code would be worked on for days at a time without being tested, and deployed sometimes years at a time. This is largely due to software historically being a physical asset

# Something changed

Two major changes have occurred over the last 10 years:

- Increased prevalence of web-based apps (no installs)
- Improvement to internet connectivity, speed, bandwidth

These changes (and more) have allowed for the pushing of updated software to **users** to be substantially more possible. Subsequently, users have come to expect more rapid updates.

**A movement from software as an asset, to software as a service,  
has catalysed this transition**

# Software as a service (Sass)



Service vs Asset

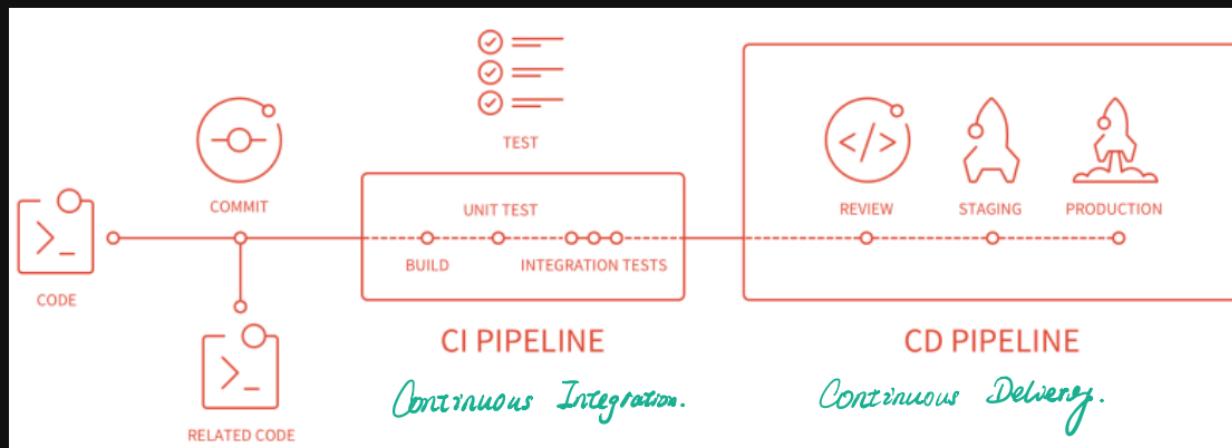
A simple case study can be found in Microsoft's movement of Windows from shipping a product, to shipping a service.

# Cloud services

- Numerous cloud services offer the ability to "easily" deploy your web applications
  - Amazon Web Services
  - Google App Engine
  - Heroku

# Modern Deployment

To achieve rapid deployment cycles, modern deployment isn't as simple as pushing code. Rather, a heavily **integrated** and **automated** approach is preferred.



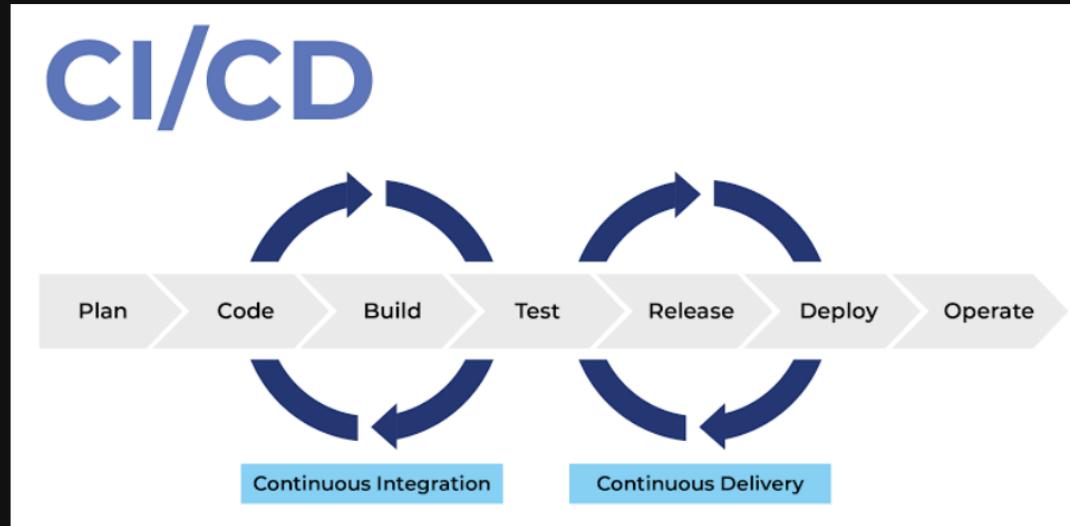
# Continuous Delivery

Continuous delivery: Allows accepted code changes to be deployed to customers quickly and sustainably. This involves the **automation of the release process such that releases can be done in a "button push".**

# Continuous Delivery

- Many companies will have a daily or weekly "ship"
- Often there is some "sign off" process before things are finally shipped
- Since the process is highly controlled, less likely to make mistakes during testing

# CI/CD relationship



# CD: Readings

- <https://www.atlassian.com/continuous-delivery/principles>
- <https://about.gitlab.com/product/continuous-integration/>

# Flighting

Continuous delivery is concerned with automatically pushing code out to dev, test, prod.

Flighting is a term used predominately in larger software projects to describe moving builds out to particular slices of users, beyond the simplicity of "dev", "test", "prod"

# Different deployments

It is common to have 3 core tiers:

- **dev:**

- released often, available to developers to see their changes in deployment

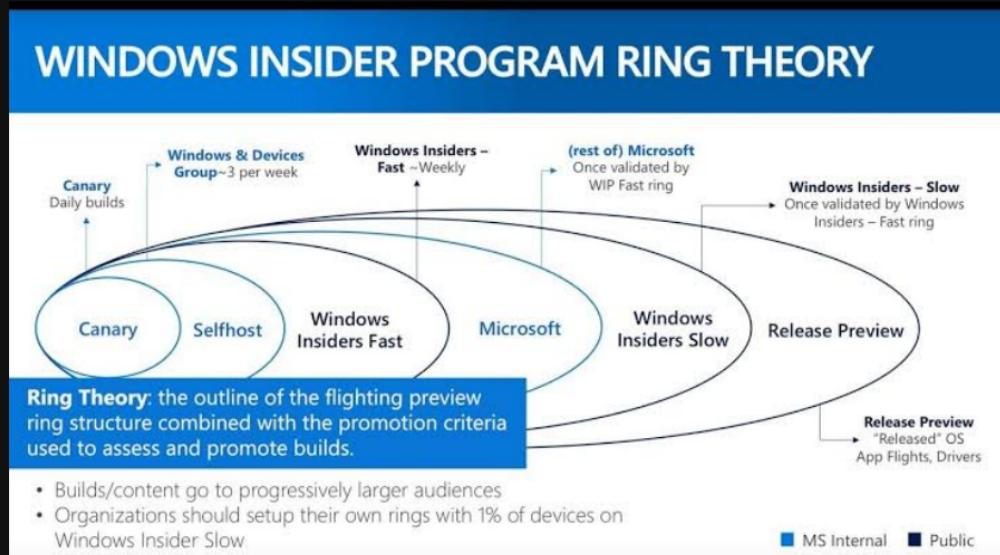
- **test:**

- As close to release as possible, ideally identical to prod

- **prod:**

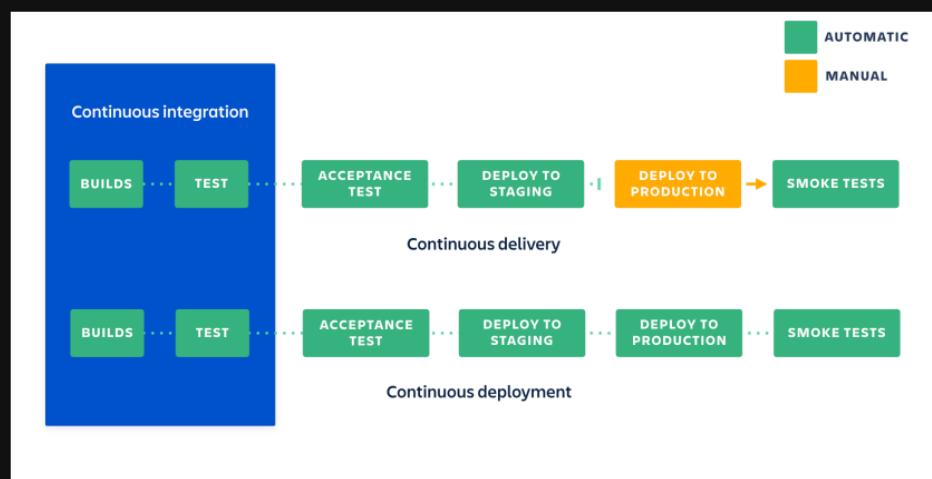
- Released to customers, ideally as quickly as possible

# Flighting



# Continuous Deployment

Continuous Deployment is an extension of Continuous Delivery whereby changes attempt to flight toward production automatically, and the only thing stopping them is a failed test



*Continuous delivery  
(Involved manual operation)*

*↓*

*Continuous deployment.  
(All automated).*

# CD: Further Reading

- <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>

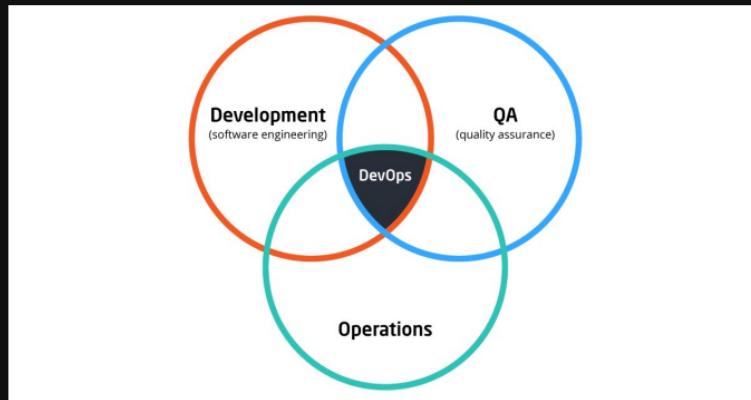
# Deploying on your own: AlwaysData

For 21T1 COMP1531 has decided to use a free service known as "alwaysdata" to let students deploy their **backend** to the cloud.

Instructions of how to set this up are found in the project repository for iteration 3. We will do a brief demo in lectures.

# DevOps

A decade ago, the notion of dev ops was quite simple. It was a role dedicated to gluing in the 3 key pillars of deploying quality assured software



*DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality [Wikipedia. Yes, Wikipedia]*

# DevOps

As development teams become less silo'ed, modern DevOps is less a role, and more a series of roles or aspect of a role.



Source & Reading: <https://hackernoon.com/devops-team-roles-and-responsibilities-6571cfb56843>

# Maintenance & Monitoring

**Maintenance:** After deployment, the use of analytics and monitoring tools to ensure that as the platform is used and remains in a healthy state.

**Monitoring often has two purposes:**

- *Preserving user experience*: Monitoring errors, warnings, and other issues that affect performance or uptime.
- *Enhancing user experience*: Using analytical tools to monitor users or understanding their interactions. Often leads to customer interviews and user stories

# Maintenance

**Maintenance:** After deployment, the use of analytics and monitoring tools to ensure that as the platform is used and remains in a healthy state.

Health is defined by developers, but often consists of:

- Monitoring 4XX and 5XX errors
- Ensuring disk, memory, cpu, and network is not overloaded

Often these aren't actively monitored, but rather monitored with alerts and triggers

# COMP1531

## 8.4 - Admin - Project - Iteration 3

Let's discuss

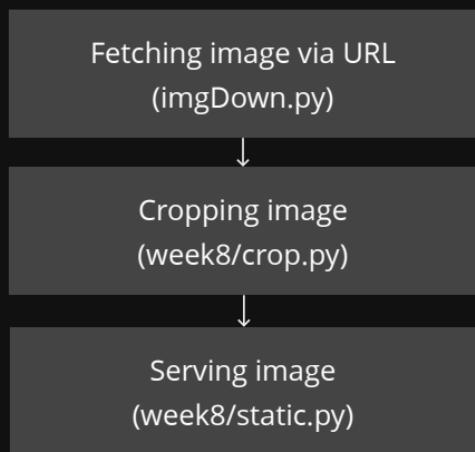
# Sending Emails

This is one of the more challenging parts of Iteration 3. You can create a dummy gmail account and make sure it is a **less** secure app.

Once that is done, you can usually search around the internet for python-based SMTP libraries that will allow you to send emails.

Work collaboratively with your team to solve this problem.

# Storing and serving images



# Using threads & timers

Most of what we do in python is single-threaded, i.e. if an action takes a while to complete, the program stops and waits for that action to be complete.

To do more interesting things like "set a timer to execute code later" we need to use the threading library so that multiple paths/streams of the program can be completed concurrently.

# Using threads & timers

timer.py

```
1 import threading
2 import time
3
4 def hello():
5     print("hello, Timer")
6
7 if __name__ == '__main__':
8     t = threading.Timer(3.0, hello)
9     t.start()
```

source:

[https://www.bogotobogo.com/python/Multithread/python\\_multithreading\\_subclassing\\_TimerObject.php](https://www.bogotobogo.com/python/Multithread/python_multithreading_subclassing_TimerObject.php)

# COMP1531

## 9.1 - SDLC Design - Software Complexity

How complicated is  
software?

# No Silver Bullet

- A famous paper from 1986:
  - *No Silver Bullet – Essence and Accident in Software Engineering* by Fred Brooks
- Described software complexity by dividing it into two categories *essential* and *accidental*.
- Further conclusions of the paper are much debated

# Essential

Complexity that is inherent to the problem.

For example, if the user or client requires the program to do 30 different things, then those 30 things are essential

# Accidental

Complexity that is **not** inherent to the problem.

For example, generating or parsing data in specific formats.

writing your own function,  
when there's corresponding library  
exist.

# Essential

Fundamentally can't be removed, but can be managed with good *software design*.

# Accidental

Can be somewhat mitigated by engineering decisions; e.g. smart use of libraries, standards, etc.

Hard to remove entirely.

# Open questions

- Is there a concrete process for distinguishing accidental and essential complexity?
- How much of the complexity of modern software is accidental?
- To what degree has or will accidental complexity be removed in future?

# Further reading

- The original No Silver Bullet paper:
  - [http://faculty.salisbury.edu/~xswang/Research/Papers  
/SERelated/no-silver-bullet.pdf](http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf)
- A more modern description:
  - [https://stevemcconnell.com/articles/software-  
engineering-principles/](https://stevemcconnell.com/articles/software-engineering-principles/)
- A recent rebuttal:
  - <https://blog.ploeh.dk/2019/07/01/yes-silver-bullet/>

Can we measure  
complexity?

Mapping the connections for all functions.

# Coupling



More links → more complicated.

We like the code more "loosely" → less link.

- A measure of how closely connected different software components are
- Usually expressed as a simple ordinal measure of "loose" or "tight"
- For example, web applications tend to have a frontend that is loosely coupled from the backend

Coupling or Cohesion.

# Cohesion

- The degree to which elements of a module belong together
- Elements belong together if they're somehow related
- Usually expressed as a simple ordinal measure of "low" or "high"



# Cyclomatic complexity

Decision :

• while loop

• if .. else statement.

- A measure of the branching complexity of functions
- Computed by counting the number of linearly-independent paths through a function

Counting the different paths you can go.

# Cyclomatic complexity

- To compute:
  1. Convert function into a control flow graph
  2. Calculate the value of the formula

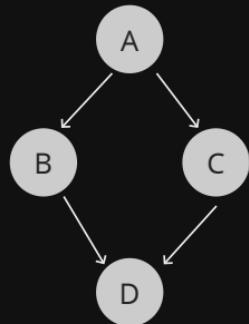
$$V(G) = e - n + 2$$

(edges)(nodes)

where e is the number of edges and n is  
the number of nodes

# Example 1

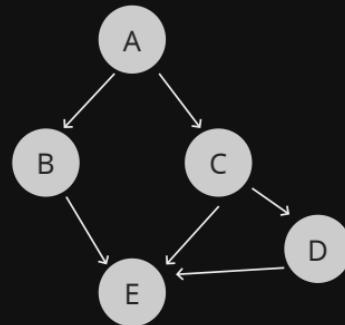
```
1 def foo():
2     if A():
3         B()
4     else:
5         C()
6     D()
```



$$V(G) = 4 - 4 + 2 = 2$$

# Example 2

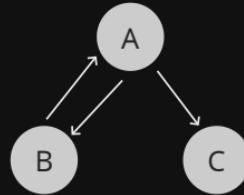
```
1 def foo():
2     if A():
3         B()
4     else:
5         if C():
6             D()
7     E()
```



$$V(G) = 6 - 5 + 2 = 3$$

# Example 3

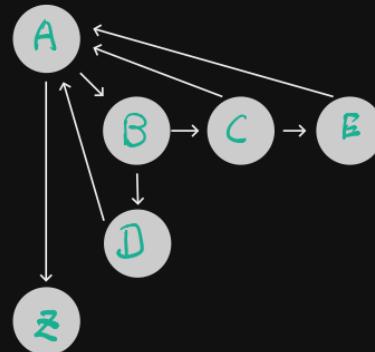
```
1 def foo():
2     while A():
3         B()
4     C()
```



$$V(G) = 3 - 3 + 2 = 2$$

# Example 4

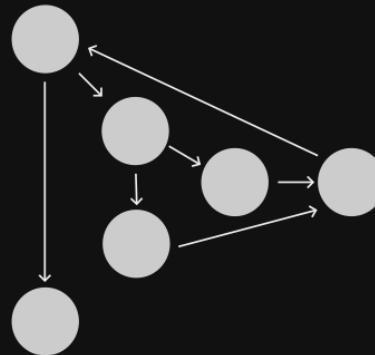
```
1 def day_to_year(days):
2     year = 1970
3
4 A: while days > 365:
5     B: if is_leap_year(year):
6         C: if days > 366:
7             days -= 366
8             year += 1
9     D: else:
10        days -= 365
11        year += 1
12
13 E: return year
```



$$V(G) = 8 - 6 + 2 = 4$$

# Example 5

```
1 def day_to_year(days):
2     year = 1970
3
4     while days > 0:
5         if is_leap_year(year):
6             days -= 366
7         else:
8             days -= 365
9         year += 1
10
11 return year - 1
```



$$V(G) = 7 - 6 + 2 = 3$$

# Usage

- A simple understandable measure of function complexity
- Some people argue 10 should be the maximum cyclomatic complexity of a function where others argue for 8

# Drawbacks

- Assumes non-branching statements have no complexity
- Keeping cyclomatic complexity low encourages splitting functions up, regardless of whether that really makes the code more understandable

# Automatic calculation

- [http://pylint.pycqa.org/en/latest/technical\\_reference/extensions.html#design-checker](http://pylint.pycqa.org/en/latest/technical_reference/extensions.html#design-checker)
- NOTE: May get different results compared to doing it by hand as the extension generates a more complex CFG

# COMP1531

## 9.2 - SDLC Development - Safety & Type Checking

# Safety

Protection from  
accidental misuse

```
number =  
String x int.
```

# Security

Protection from  
deliberate misuse

# Case study: spreadsheets

- Around 94% of spreadsheets contain errors\*
- For any given spreadsheet formula, there's a 1% chance it contains an error\*\*
- Why?

\* What We Know About Spreadsheet Errors (2005)

\*\* Errors in Operational Spreadsheets (2009)

# Software safety

- Things that can go wrong:
  - C:
    - Reading from memory that has not been initialised
    - Dereferencing a null pointer
    - "Using" memory after it has been freed
    - Writing outside the bounds of an array
    - Forgetting to free allocated memory
  - Python:
    - Accessing a variable that hasn't been initialised
    - Accessing a member that an object doesn't have
    - Passing a function a type of object it doesn't expect

*we prefer*

*static.*

## Static

Static properties can be inferred without executing the code

E.g. pylint statically checks that variables are initialised before they're used

*compile → execution*

*we want  
error can be  
checked during  
compile time.*

## Dynamic

Dynamic properties are checked during execution

E.g. python dynamically checks that an index is inside the bounds of a list and throws an exception if it isn't (unlike an array in C)

*Raise exemption...*

# Memory safety

- Protecting from bugs relating to memory access
- Python is memory safe as it prevents access memory that hasn't been initialised or allocated
- The checks are mostly dynamic (at runtime)
- In python, safety is prioritised over the *negligible* performance cost of bounds-checking

Array [1, 2, 3, 4]

# Memory Safety

Array [5] = ...

→ C would not check.

→ python check the length and give an error.  
time (performance) cost.

- C is not memory safe
- No bounds checking is performed for array accesses
- Pointers can still be dereferenced even if they don't point to allocated memory
- C prioritises performance over safety (and security)

# Handling runtime errors

- Different languages have difference conventions for handling errors
- Python relies on Exceptions for the majority of error handling. E.g.  
1 `animals["fish"]`  
will throw a `KeyError` exception if "fish" is not in the dictionary `animals`.
- C does not support exceptions at all, so errors typically have to be included in the return value.

# Easier to Ask for Forgiveness than Permission

Python: `try .. except ..`

*continues*

C : `Assert(....)`  
`exit Error ..`

*cannot continue ..*

- EAFP is the python convention for handling errors.
- It encourages you to assume something will work and just have an exception handler to deal with anything that might go wrong
- Pros:
  - Can simplify the core logic
  - Multiple different sorts of errors can be handled with one except block
- Cons:
  - Makes code non-structured
  - Harder to reason what code will be executed.

# Look Before You Leap

- **LBYL** is a convention for avoiding errors popular in languages like C
- Unlike EAFP it encourages you to check that something can be done before you do it
- Pros:
  - Doesn't require exceptions
  - Code is structured and therefore easier to reason about
- Cons:
  - Core logic can be obscured by error checks

# Removing errors statically

- Rather than dynamically checking for certain errors, it is always better if errors can be detected statically
- Rules out entire classes of bugs
- In Python, pylint can statically detect certain errors (e.g. unknown identifier)
- In C, the compiler detects a number of errors including type errors.

# Type safety

- Preventing mismatches between the actual and expected type of variables, constants and functions
- C is type-safe\*, as types must be declared and the compiler will check that the types are correct
- Python, on its own, is not type-safe. Everything has a type, but that type is not known till the program is executed

\* mostly

# Type-checking

- Languages with a non-optional built-in static type checking
  - C
  - Java
  - Haskell
- Languages with optional but still built-in static type checking
  - Typescript
  - Objective C
- Languages with optional external type checkers
  - Python
  - Ruby

# Mypy

- Mypy is a type checker for python
- Python allows you to give variables static types, but without an external checker they are ignored
- Because of python's semantics, type checking it can be complex
  - Duck typing
  - Objects with dynamically changing members

# Examples

```
1 def count(needle, haystack):
2     """
3         Returns the number of copies of integer needle in the list of integers haystack.
4     """
5     copies = 0
6     for value in haystack:
7         if needle == value:
8             copies += 1
9     return copies
10
11 def search(needle, haystack):
12     """
13         Returns the first index of the integer needle in the list of integers haystack.
14     """
15     for i in range(len(haystack)):
16         if haystack[i] == needle:
17             return i
```

# Further reading

- The Mypy website:
  - <http://mypy-lang.org/>
- How Dropbox uses MyPy
  - <https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/>

# COMP1531

## 9.3 - SDLC Testing - Property-based Testing

# Testing: what's the problem?

- Good testing gives us strong guarantees about our software, but...
  - Tests take a lot of time to write
  - It's easy to miss edge cases (even with coverage checking)
  - These problems only get worse with larger and more complicated systems

# Can the machines do it for us?

- To write a program that generates tests for us requires:
  - A means of generating test data
  - Knowing what behaviour is correct?
  - Some way for the computer to report test failures to us in an understandable way

# Example: Bubblesort

```
1 def bubblesort(numbers):
2     numbers = numbers.copy()
3     for _ in range(len(numbers) - 1):
4         for i in range(len(numbers) - 1):
5             if numbers[i] > numbers[i+1]:
6                 numbers[i], numbers[i+1] = numbers[i+1], numbers[i]
7     return numbers
```

# Property-based testing

- A method of testing where tests are defined as general properties (i.e. parameterised predicates)
- Test input is generated automatically by supplying a strategy for generating that input
- The testing framework runs the test many times to ensure the properties are true for each input
- In the event of a test failure, the framework will shrink the generated input to find the smallest value that still fails the test

# Hypothesis

- Hypothesis is the name of property-based testing framework for python
- Can be installed via: **pip3 hypothesis**
- Parameterised tests are decorated with @given to supply strategies for generating test input
- See **bubblesort.py**

# What properties to test?

- It's not always easy to find testable properties
- Can you think of one for the Zune bug example?
- Software designs with testable properties tend to be good designs...

# COMP1531

## 10.1 - Week 10 General

# Iterators & Generators

- Evan Kohilas gave a more extensive talk on this topic during a CSESoC supported event a couple of weeks ago - but we will cover them lightly today

# Iterators

- In Python, iterators are objects *containing* a countable number of elements
- For example, we can get an iterator for a list:

```
1 animals = [ "dog", "cat", "chicken", "sheep" ]  
2  
3 animal_iterator = iter(animals)
```

# Iterators

- Any object with the methods `__iter__()` and `__next__()` is an iterator
- Simple example (squares)

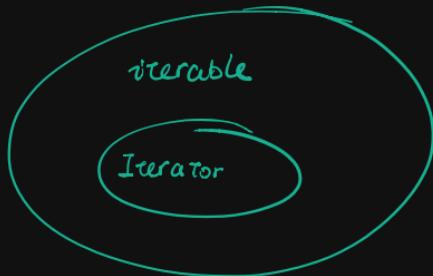
```
1 class Squares:  
2     def __init__(self):  
3         self.i = 0  
4  
5     def __iter__(self):  
6         return self  
7  
8     def __next__(self):  
9         self.i += 1  
10        return self.i*self.i
```

# For loops

- Python for loops use iterators behind the scenes
- This is valid code:

```
1 squares = Squares( )
2
3 for i in squares: # Loops forever
4     print(i)
```

# Iterator vs Iterable



- Intuitively:
  - An iterator stores the state of the iteration (i.e. where it's up to).
  - Something is iterable if it can be iterated over.
- Concretely:
  - An iterator has `__iter__()` and `__next__()` methods.
  - Iterables have `__iter__()` methods
- Thus, all iterators are iterable, but not all iterables are iterators
  - For example, lists are iterable, but they are not iterators
  - For loops only need to be given something *iterable*

# Generators

- A different way of writing iterators
- Defined via generator functions instead of classes
- Example generator

```
1 def simple_generator():
2     print("Hello")
3     yield 1
4     print("Nice to meet you")
5     yield 2
6     print("I am a generator")
```

# Generators

- Intuitively, you can think of a generator as a suspendable computation
- Calling `next()` on a generator executes it until it reaches a `yield`, at which point it is suspended (frozen) until the subsequent call to `next()`

# Generators

- More useful examples

```
1 def squares():
2     i = 0
3     while True:
4         i += 1
5         yield i*i
```

```
1 def fib():
2     a = 1
3     b = 1
4     while True:
```

# Libraries

- Most code re-use is through libraries.
- Software engineering can be an exercise in composing libraries to do what we want.
- This is necessary for building *useful* software.
- What's the downside?

# Case study: leftpad

- A Javascript library that had many users, mostly indirect
- Owing to a disagreement, the author removed the library from NPM
- This caused thousands of Javascript-based applications and libraries to break

# The entire library

```
1 module.exports = leftpad;
2 function leftpad (str, len, ch) {
3     str = String(str);
4     var i = -1;
5     if (!ch && ch !== 0) ch = ' ';
6     len = len - str.length;
7     while (++i < len) {
8         str = ch + str;
9     }
10    return str;
11 }
```

# Further reading

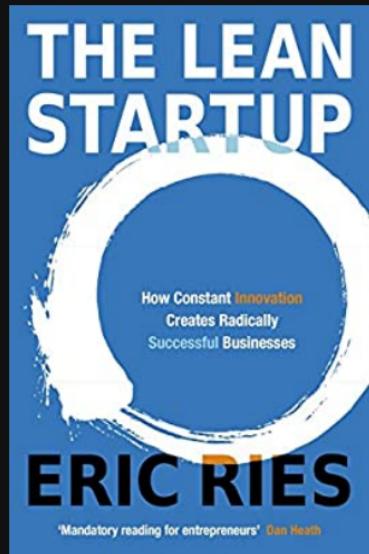
- An analysis of the leftpad incident
  - <https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/>
- Dependency Hell
  - [https://en.wikipedia.org/wiki/Dependency\\_hell](https://en.wikipedia.org/wiki/Dependency_hell)
- An attempt fix to dependency hell
  - <https://nixos.org/nix/>

# COMP1531

## 10.2 - Building an MVP

# Being Lean

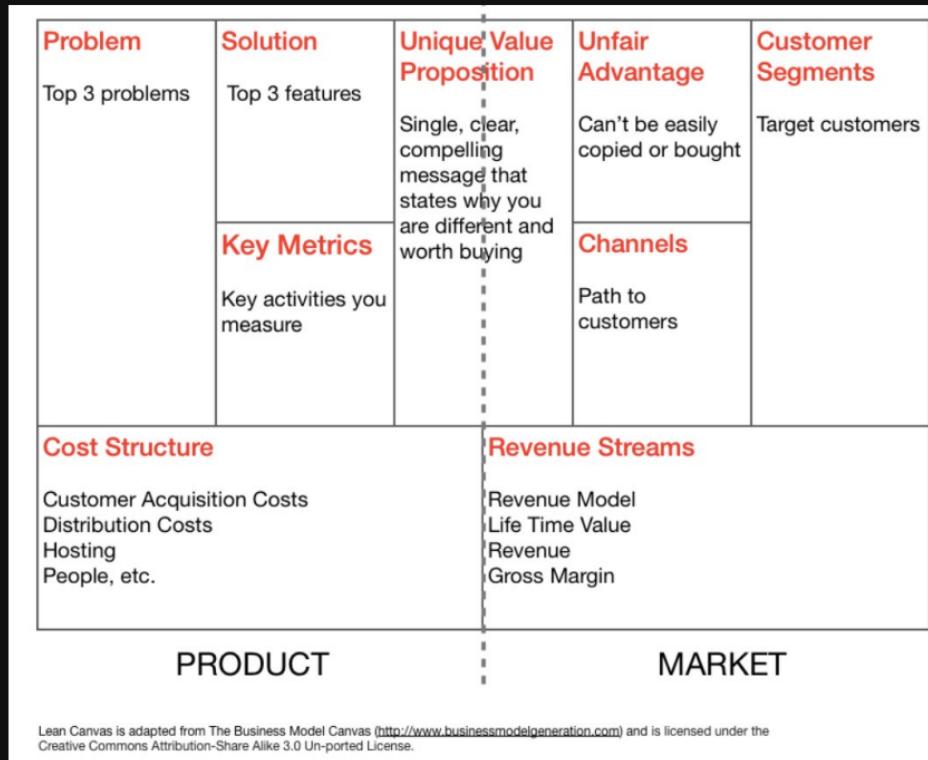
- Eric Ries produced a very popular book known as the "Lean Startup". The book focuses on methods to shorten the product development life cycle and determine if a business is viable or not more rapidly.
- "Build-measure-learn" principles



# Lean Canvas

- A "Lean Canvas" is a common tool used to try and articulate the core principles of a business
- These canvases can then be used to inform what to build.
- If you ever want to start a business - start here

# Lean Canvas



# Product-Market Fit

- Product-Market Fit (PMF) is the degree to which a product satisfies a strong market demand.
- Young companies thrive to reach PMF as soon as possible, as it's usually a key step in early growth
- One of the earliest steps of exploring PMF in a lean way, and verifying the business viability, is to build an **MVP**
  - To build an MVP, one must first understand the business
  - The sole purpose of an MVP is to validate assumptions, not to build good technology