COMP1531 – Course Notes

Contents

Week 1

Software Engineering: 1.1 – Overview

- Goal of the course: Learning software engineering by programming in teams to build user-facing applications by applying the software development lifecycle

What is a software engineer?

- 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"

Why have software engineering?

- 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

Software Development Life Cycle (SDLC)

1. Requirements Analysis
    - Understand the problem you are trying to solve
    - Analyse and understand the problem domain
    - Determine functional and non-functional components
    - Generate user stories / use cases
2. Design
    - Producing software architecture/blue-prints
    - System diagrams and schematics
    - Modelling of data flows
    - Happens ideally before you write any code
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
5. Deployment
    - Make the software available for use by the users
6. Maintenance
    - Monitor the system, track issues, interview users to reassess requirements (start again!)

Teamwork: 1.2 – Git – Solo Usage

- Need for a method of managing our code:
    - Version control: Track changes in our code
    - Concurrent programming: Allow multiple people to work on the same files or series of files and seamlessly integrate changes together

Git

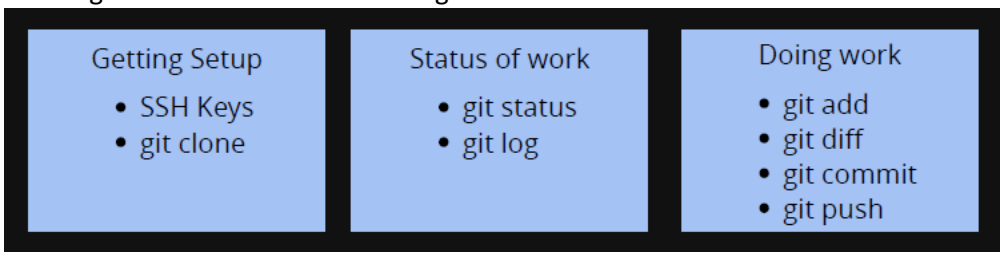
- Use Git!
- Git is a distributed version control software
- Whilst many users share work via a central cloud, each user has a full copy of the work and therefore each user has a full backup of the work

Git Tools

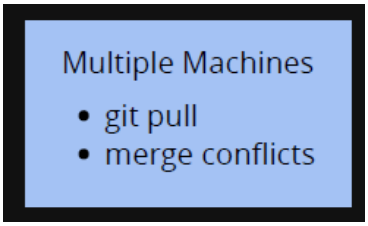
- Git is just a programming language, but we use tools to interact with it:
    - Github
    - Bitbucket
    - Gitlab

Git – Single Machine

- Stage 1: Version control on a single machine



- Stage 2: Version control across multiple of your machines



- Some git commands

Command	Description	Example
git clone	Clones from a cloud repository to a local repository	git clone
git status	Tells you information about the "state" of your repo	git clone
git log	Gives you a commit history of commits made	git clone
git add	Adds a particular untracked file to your repo ready for commit, or stages a tracked file ready for commit	git add --all git add file.py
git diff	Shows the difference between the last commit and the work you've done since then	git diff
git commit	Commits changes ("takes a snapshot") of your work	git commit -m "Message name"
git push	Syncs the commit history locally with the commit history on the cloud	git push git push origin master
git pull	Syncs the commit history on the cloud with the commit history locally	git pull git pull origin master

Python: 1.3 – Introduction

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

Other Languages

	Procedural	Object-oriented	Typed	Pointers	Compiled
C	Yes	No	Yes	Yes	Yes
C++	Yes	Yes	Yes	No	Yes
Java	No	Yes	Yes	No	Yes
Python	Yes	Yes	Can be	No	No
Javascript	Yes	Yes	Can be	No	No

Main differences with Python and C

- No types
- Has object-orientated components
- Does not have pointers
- Written at a 'higher level'
- Does not have an intermediate compilation step

Mutable and Immutable

- To summarise the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.

Strings

- Strings are immutable

Lists and Tuples

- Lists are for mutable ordered structures of the same type
- Tuples are for immutable ordered structures of any mix of types

Slicing

- Lists/Tuples can be "sliced" to extract a subset of information about them

```
chars = ['a', 'b', 'c', 'd', 'e']

## Normal Array/List stuff
print(chars)
print(chars[0])
print(chars[4])

## Negative Indexes
print(chars[-1])

## Array Slicing
print(chars[0:1])
print(chars[0:2])
print(chars[0:3])
print(chars[0:4])
print(chars[0:5])
print(chars[2:4])
print(chars[3:5])
print(chars[0:15])
print(chars[-2:-4])
```

Control Structures

```
# Note the following:
#  - Indentation and colon denotes nesting, not braces
#  - Conditions generally lack paranthesis
#  - pass used to say "do nothing"
#  - i++ is not a language feature

number = 5
if number > 10:
        print("Bigger than 10")
elif number < 2:
        pass
else:
        print("Number between 2 and 9")

print("--------------------------")

i = 0
while i < 5:
        print("Hello there")
        i += 1

print("--------------------------")
for i in range(5):
        print("Hello there")
```

Functions

```
def get_even(nums):
        evens = []
        for i in range(len(nums)):
                if number % 2 == 0:
                        evens.append(number)
        return evens

all_numbers = [1,2,3,4,5,6,7,8,9,10]
print(get_even(all_numbers))
```

Dictionaries

- Have a key and value pair

```
student = {
    'name': 'Emily',
    'score': 99,
    'rank': 1,
}

print(student)
print(student['name'])
print(student['score'])
print(student['rank'])

student['height'] = 159
print(student)
```

Combining

- Create data structures of other structures

```python
student1 = { 'name' : 'Hayden', 'score': 50 }
student2 = { 'name' : 'Nick', 'score': 91 }
student3 = { 'name' : 'Emily', 'score': 99 }
students = [student1, student2, student3]

print(students)

# Approach 1
num_students = len(students)
for i in range(num_students):
  student = students[i]
  if student['score'] >= 85:
    print(f"{student['name']} got an HD")

# Approach 2
for student in students:
  if student['score'] >= 85:
    print(f"{student['name']} got an HD")
```

Software Engineering: 1.4 – Testing - Intro

- We need to make sure our application works properly
- Printing errors or visually inspecting the out is a method of debugging not testing.
- You cannot call something a testing method if it doesn't scale well.

Assert

- Built in function assert will raise an error if what it is provided is not true

Blackbox Testing and Abstraction

- Abstraction is the notion of focusing on a higher-level understanding of the problem and not worrying about the underlying detail
- We do this when diving a car
- When we look at systems in an abstract way, we could also say that we're treating them like black boxes

Design by Contract

- When we are testing or implementing a function, we will typically be working with information that tells us the constraints placed on at least the inputs
- This information tells us what we do and don't need to worry about when writing tests

Pytest

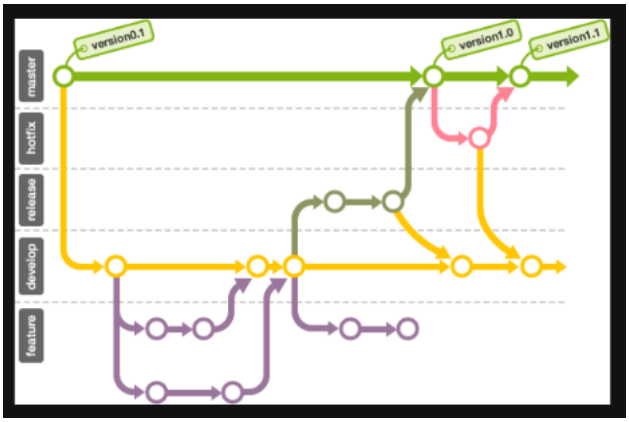
- Pytest is a structured method of writing, organising, and running tests
- It comes with a binary that we run on command line
- Pytest detects any function prefix with test and runs that function, processing the assertions inside

Week 2

Teamwork: 2.1 – Git – Team Usage

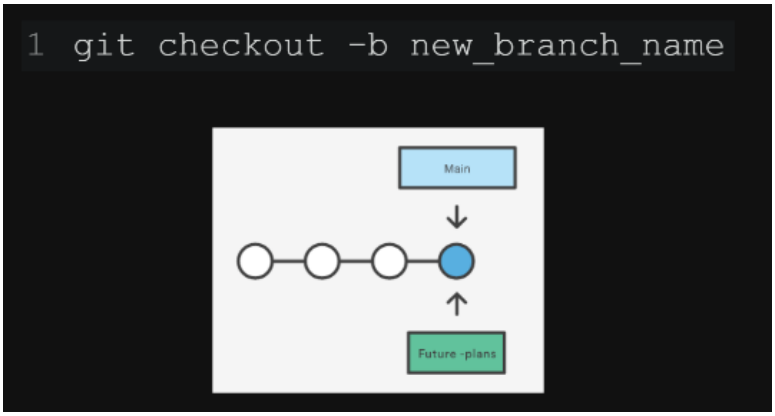
The Git Tree Model

- Git can be understood as a tree-like structure
- Git is a collection of commits
- Each commit has one parent.
- Each commit can have multiple children (i.e. branches)
- A branch essentially is just a pointer to a particular commit
- To try to bring two separate branches together onto the same commit is a process of 'merging'



Branches

- The master branch is just a pointer to a particular commit on master (usually the latest)
- You can create your own branch if you want to continue on a separate thread of working, unrelated to the master branch



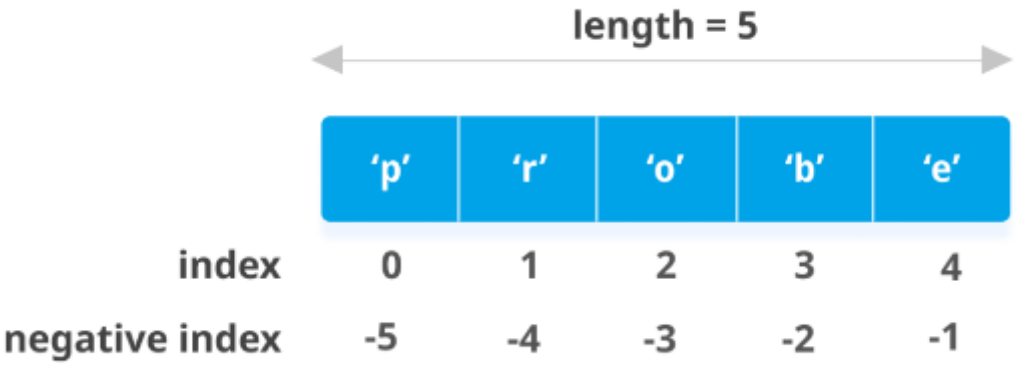
Merging

- Merging is the process of 'incorporating work on another branch in mine'/
- Usually, you will be merging master into your work whilst you develop on it
- Merging your work into master once your branch is stable enough to merge into master
- The merge command let's you specify the branch you want merged into your current branch, i.e. git merge master

Python: 2.2 – Collections

Lists

- Lists are sequential containers of memory



Tuples

- Syntactically, tuples are similar to lists except:
  - They are read-only once created
  - You create them with () instead of []
- In terms of usage, tuples tend to be used more often where:
  - You have a collection of different types
  - The collection is of fixed size

Dictionaries

- Dictionaries are associative containers of memory
- Values are referenced by their string key that maps to a value
- In newer versions of python, dictionary keys do have a sense of order

Sets

- Sets are essentially unordered collection of keys (with no associated values)
- They are like lists but with {}

Python: 2.3 – Importing & Paths

if __name__ == '__main__':

- Will only run if the file is being run
- Won't be run if it is imported to another file

Importing

- Methods to import:
    - Method 1 pollutes the name space (imports everything like method 2): from lib import *
    - Method 2 generally clearest: from lib import one, two, three
    - Method 3 useful if imported items need context (lib.one() to do something): import lib
- When you import in python, python will look for that module:
    1. As one of the built-in binaries
    2. Then, relative to the directory your python process was invoked in
    3. Then, relative to directories in the sys.path list
- For (2), without thinking more deeply about it, this is why it's important in a project you try and invoke files from a consistent directory

Python: 2.4 – Packages and Virtual Environments

Importing Libraries

- Python comes packaged with a number of standard libraries (i.e. 'math')
- However, many libraries that you may want to use have to be installed for usage
- Installing extra libraries can be made easy with the pip program
- Pip installs a particular version of a particular python module to your system
- The version for pip used for python3 is known as pip3

Installing with pip

- Where is it stored?
- Typically in your home directory

Potential Issues with Installing

- 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

- A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them
- Read more here: https://www.geeksforgeeks.org/python-virtual-environment/ and https://realpython.com/python-virtual-environments-a-primer/

```
pip3 install virtualenv
python3 -m virtualenv venv/
source venv/bin/activate

# Do stuff

pip3 freeze > requirements.txt # Save modules
pip3 install -r requirements.txt # Install modules

deactivate
```

Packages

- In conclusion, non user-defined packages can be found in 1 of 4 places:
    - Built-in to python (no action needed)
    - Installed on the system (sudo pip3 install [x]) – in sys.path
    - Installed in your home directory (pip3 install [x]) – in sys.path
    - Installed in your project folder (pip3 install [x] within venv)
- Activating a virtual environment means we no longer look in (3) and instead look in (4)

Python: 2.5 – Exceptions

What is an exception?

- 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
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    inputNum = int(sys.stdin.readline())
    print(sqrt(inputNum))
```

Try and Except Statements

- Test a block of code in try, if it raises the exception then execute Exception part
- The 'as e' part captures the error message

```python
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    try:
        print("Please enter a number: ",)
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
    except Exception as e:
        print(f"Error when inputting! {e}. Please try again:")
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
```

Pytest

```python
import pytest

def sqrt(x):
    if x < 0:
        raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
    return x**0.5

def test_sqrt_ok():
    assert sqrt(1) == 1
    assert sqrt(4) == 2
    assert sqrt(9) == 3
    assert sqrt(16) == 4

def test_sqrt_bad():
    with pytest.raises(Exception):
        sqrt(-1)
        sqrt(-2)
        sqrt(-3)
        sqrt(-4)
        sqrt(-5)
```

Teamwork: 2.6 – Interactions and Operations

What is agile?

- Agile is an iterative approach to project management and software development that helps teams deliver value to their customers faster and with fewer headaches

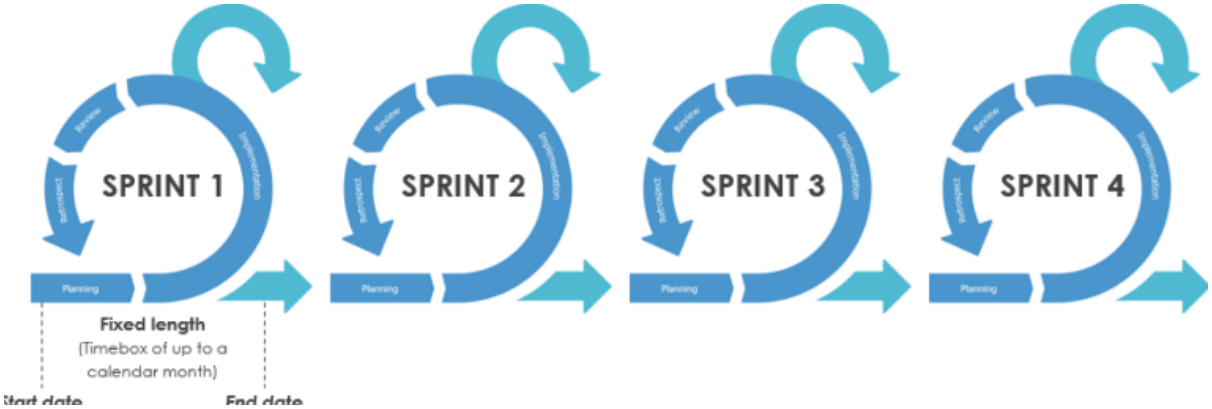
Standups

- 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?

Task Boards

Sprints

- A fixed time (e.g. week, fortnight) where you set a number of tasks to be completed in the team
- After that period is up, you review progress, and set tasks for the next sprint
- Time is fixed, scope is flexible
- Plan only for the next sprint
- Typically have a release at the end of each sprint



Meeting Minutes

- It's a good idea for teams to have more formal meetings at least once a fornight
- Typically this kind of meeting would fall into a start-of-sprint style meeting, though in the absence of sprints just a 'weekly meeting' is adequate
- During meeting it is usually good to have someone take meeting minutes (i.e. 'notes')
- Meeting minutes will typically consist of documenting:
    - Attendees
    - (Optional) Agenda
    - Discussion points
    - Actions

Pair Programming

- Two programmers, one computer, one keyboard
- Take 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

Week 3

Python: 3.1 – Objects

Various Types in C

- Simple types in C were basic types that occupied limited memory
- Structs were collections of primitive types wrapped into an 'object'
    - We would create instances of these 'objects' and then access properties of them
    - We can expand this concept into Python

Python Objects

- In python, basically every data type acts like an 'object'
- An 'object' being a data type that:
    - Can be created via a constructor
    - Contains 0 or more properties (/attributes)
    - Contains 0 or more functions (/methods)
- To oversimplify: It's structs with functions

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

Everything* is an object

- Almost all values in python are objects
- We refer to object types as 'classes'
- For example:
    - Lists have an append() method
    - Strings have a capitalize() method

Classes

- Classes allow you create your own classes (object types) though in this course you won't need to

Software Engineering: 3.2 – Testing - Linting

Good Style

- Good style means that:
    - It is easier to follow the flow of code with consistent whitespace
    - It is easier to visually glance at code with similar patterns
    - It can be easier to detect bugs
        - E.g. if you force constants to be uppercase name, it's easy to spot a mutated uppercase variable
- Be consistent with style, if you start with snake case, stick with snake case!

Who decides on styles?

- Choose a style guide set by a large organisation (e.g. Google, Microsoft, Facebook, etc)
- (Optional) Modify specific style rules to satisfy any clear subjective options (e.g. some companies believes in spaces over tabs)
-

What is 'static analysis'

- Static analysis is the process of analysing as much of your program as you can before running it
- E.g. C compilation contains elements of static analysis (e.g. type checking, unused variables, etc)
- Linting python code consist of:
    - Style issues (whitespace, indentation)
    - Semantic issues (bad logic, potential bugs)
- Because python is interpreted (no compile step) linting helps bridge the gap of some things missed out by compilers

Pylint

- A popular external tool used to statically analysing python code
- Can detect errors, warn of potential errors, check against conventions, and give possible refactoring suggestions
- By default, it is very strict
- Can be configured to be more lenient through configuration

Fixing Pylint Issues

- Semantic issues often need to be fixed manually
- Style issues can be fixed either via:
    - Using IDE addons, such as auto-formatters for VSCode
    - Using tools like autopep8 to parse your code

- 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
```

Software Engineering: 3.3 – Testing – Continuous Integration

- To scale multi-user software projects we need to mated ways to integrate and test code

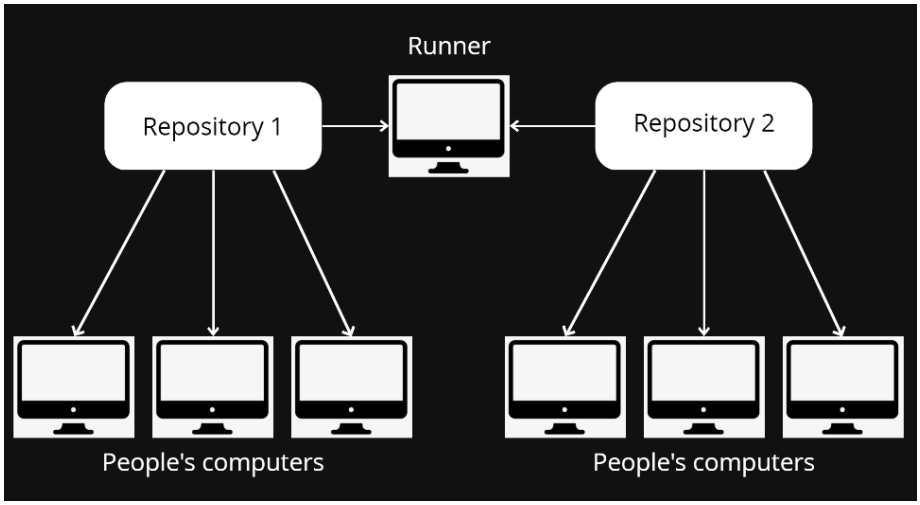
Continuous Integration (CI)

- Practice of automating the integration of code changes from multiple contributors into a single software project
- Key principles and processes:
    - Write tests:
        - Ideally tests for each story
        - Broad tests: unit, integration, acceptance, UI tests
    - Use code coverage checkers
    - Merge and integrate code as often as possible
    - Ensure the build always work

How it works

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

Broad Architecture



Continuous Integration, gitlab

- Gitlab, like many source control tools, has a way of doing continuous integration.
- In gitlab repos, you can setup your own conitnous integration if:
    - You connect a runner to your repository
    - You setup a .gitlab-ci.yml file

Software Engineering: 3.4 – Verification and 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 the 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

Verification Types

- Verification can be broken up into two main types:
    - Static verification (what you know as 'linting')
    - Dynamic verification (what you know as 'testing')
- Static verification is usually considered a more robust and reliable form of testing. However, in many cases we need to dynamically verify code too

Static Verification

- Methods of static verification include:
    - (Linting) Style checking
    - (Linting) Type checking
    - (Linting) Logic checking. Includes:
        - Anti-pattern detection
        - Potential warnings
    - Key metric checking. Includes:
        - Coupling
        - Cyclomatic complexity
    - Formal verification
    - Informal verification

Dynamic Verification

- Verification performed during the execution of software
- Often known as the 'test' phrase
- Typically falls into one of three categories
    - Testing in the small
    - Testing in the large
    - Acceptance tests

Dynamic – Testing in the small

- We often refer to small tests as unit tests, which the ISTQB (International Software Testing Qualifications Board) defines as the testing of individual software components
- These can be white-box or black-box tests, and are written often by engineers who will implement work

Dynamic – Testing in the large

- Larger tests are tests performed to expose defects in the interfaces and in the interactions between integrated components or systems (ISTQB definition).
- These tests tend to be black-box tests and are written by either developers or independent testers
- Typically these tests fall into these categories:
    - Module tests (testing specific module)
    - Integration tests (testing the integration of modules)
    - System tests (testing the entire system)

Dynamic – Acceptance Testing

- ISTQB defines acceptance testing as formal testing with respect to user needs, requirements, and business processes conducted tot determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorised entity to determine whether or not to accept the system
- This testing always black-box, and is typically testing on the customer or user themselves
- Can either be performance based (Trying to reach specific outcome) or stress testing

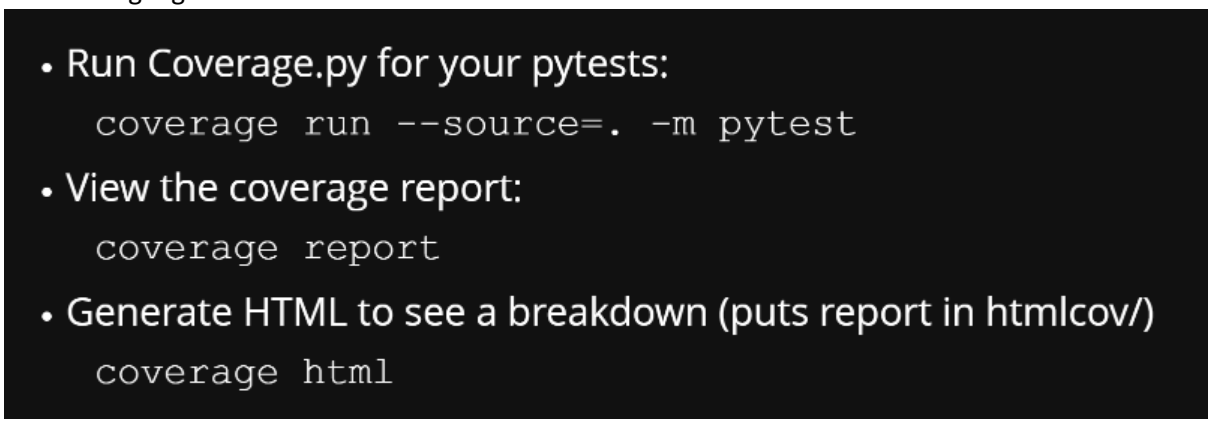
Software Engineering: 3.5 – Testing – Code Coverage

Coverage

- Test Coverage: A measure of how much of the feature set is covered with tests.
    - This is often left to human judgement
- Code coverage: A measure of how much code is executed during testing
    - This can be computed and quantified

Coverage.py

- Measure code coverage as a percentage of statements (lines) executed
- Can give us a good indication of how much o four code is executed by the tests
- And highlight which has not been executed



Branch coverage checking

- For lines that can potentially jump or 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

Week 4

Software Engineering: 4.1 – Development – Data Transfer

- Need for a common way to communicate between different operating systems, servers and programming languages

Standard Interfaces

- In any field of 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

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
- The 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 – Writing & Reading

- Python has powerful built in libraries to write and read JSON
- This involves converting JSON between 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 to it's ease of editing and concise nature.
- It's easy to convert between JSON and YAML online

```yaml
---
locations:
- suburb: Kensington
  postcode: 2033
- suburb: Mascot
  postcode: 2020
- suburb: Sydney CBD
  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
- Issues:
  - More verbose (harder to read at a glance)
  - More demanding to process/interpret
  - More bytes required to store (due to open/closing tags)
- Many legacy systems will still use XML as a means of storing data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <locations>
        <element>
            <postcode>2033</postcode>
            <suburb>Kensington</suburb>
        </element>
        <element>
            <postcode>2020</postcode>
            <suburb>Mascot</suburb>
        </element>
        <element>
            <postcode>2000</postcode>
            <suburb>Sydney CBD</suburb>
        </element>
    </locations>
</root>
```

Web: 4.2 – HTTP & Flask

Computer Networks

- A group of interconnected computers that can communicate
- A LAN (Local Area Network)
- An intranet
- The internet

The Internet

- A single, large, global network

World Wide Web

- A combination of three technologies: HTML, URL and HTTP
- All served over a network (the internet).

Flask

- Lightweight HTTP web server built in python

```python
from flask import Flask
APP = Flask(__name__)

@APP.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    APP.run()
```

```
$ python3 flask1.py
```

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

RESTful APIs

Method	Operation
POST	**C**reate
GET	**R**ead
PUT	**U**pdate
DELETE	**D**elete

Talking to Flask

- How can we talk to flask?
    - API client
    - Web Browser
    - URLLib via python

Web: 4.3 – HTTP Testing

Wrapping Iteration 1

```
def channels_listall_v1():
    return datastore.get_all_channels()
```



```
from json import dumps
from flask import Flask, request
from search import search_fn
APP = Flask(__name__)

@APP.route('/channels', methods=['GET'])
def search_channels():
    # check auth
    return dumps(channels_listall_v1())

if __name__ == '__main__':
    APP.run()
```

Week 5

Software Engineering: 5.1 – Design – Writing Good Software

Writing Good Software

- Generally speaking, we can consider software well written if:
    - (Testing) It's correctness is verifiable in an automated way
    - (Design) It is planned, modular, and resistant to breaking changes
    - (Development) It is clean and easy to work with

Well Designed Software

- Something happens between writing tests and finishing code: Design
- The design of software happens when you know what problem you're trying to solve with code, but what to think about the best way to solve the problem before you finish the code

Design: Thoughtful planning

- Thinking hard before you code is a great tactic to ensure that you time coding is efficient and that you carry useful documentation
- Examples:
    - Writing pseudocode
    - Flow diagrams
    - Component diagrams
    - State diagrams

Design: Robust coding

- Just because you plan software well, doesn't mean when you come to write it that it will turn out well designed in its execution

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'
- Why do we care?
    - When you repeat yourself less, a change in one module is unlikely to break entire systems

KISS (Keep It Simple Stupid)

- KISS principles state that a software system works best when things are kept simple
- It is the believe that complexity and errors are correlated
- Your aim should often be to use the simplest tools to solve a problem in a simplest way
- 'Every line of code you don't write is bug free'
- Why do we care?
    - Complicating things with boutique solutions means more code to maintain (more likely to break)
    - More code written means more code to test

Minimal Coupling

- Coupling is the free of interdependence between software components
- The more software components are connected, the more changes and alterations to one component may break another (either at compile time or runtime)
- Excessive coupling can lead to spaghetti code

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
- Why do we care?
    - Removes unnecessary code, less to maintain, less likely to cause problems

Refactoring

- Restructuring existing code without changing its external behaviour
- Typically this is to fix code or design smells and thus main code more maintainable
- Code smell: Changing in one place of code, causes you to change in more places of code

Well Developed Software

- Software can be correct (tested), well designed, but still written like absolute garbage. We want to write nice code, because:
    - It's easier for future you to read and understand
    - It's easier for others to read and understand
    - It's easier to find and spot errors now and in future
- "Clean code' is often a language specific topic
- Some things we talk about are universal
- Some things only apply to a handful of languages
- Some things apply to python specifically

Being Pythonic

- Within python, being 'Pythonic' means that your code generally follows a set of idioms agreed upon by the broader python community

Docstrings

- Docstrings are an important way to document code and make it clear to other programmers the intent and meaning behind what you're writing
- We are somewhat different on the formatting, but we want to include description, parameters and returns

Map, Reduce, Filter

- These functions help accomplish basic iterative tasks without the overhead of a loop setup
- Map: creates a new list with the results of called a provided function on every element in the given list, returns map object

```
def myfunc(a):
  return len(a)

x = map(myfunc, ('apple', 'banana', 'cherry'))

print(x)

#convert the map into a list, for readability:
print(list(x))
```

```
<map object at 0x2b4e815e7130>
[5, 6, 6]
```

- Reduce: Executes a reducer function (that you provide) on each member of the array result in a single output value

```
from functools import reduce

def do_sum(x1, x2):
    return x1 + x2

print(reduce(do_sum, [1, 2, 3, 4]))
```

```
(((1 + 2) + 3) + 4) => 10
```

```
# python code to demonstrate working of reduce()

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2, ]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))
```

The sum of the list elements is : 17

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

The `filter()` function extracts elements from an iterable (list, tuple etc.) for which a function returns `True`.

Example

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# returns True if number is even
def check_even(number):
    if number % 2 == 0:
          return True

    return False

# Extract elements from the numbers list for which check_even() returns True
even_numbers_iterator = filter(check_even, numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)

# Output: [2, 4, 6, 8, 10]
```

Exceptions > Error Codes

- C-style programming follows a principle of methodical process of using return values to denote particular errors
- Whilst this makes programs more easy to reason with, it convolutes code and makes it hard to understand. For this reason, we prefer exceptions

Multi-line Strings

```
if __name__ == '__main__':
    text1 = """hi

    this has lots of space

    between chunks"""

    text2 = (
        "This is how you can break strings "
        "into multiple lines "
        "without needing to combine them manually"
    )

    print(text1)
    print(text2)
```

Web: 5.2 – HTTP – Authentication and Authorisation

Auth vs Auth

- Authentication: Process of verifying the identity of a user
- Authorisation: Process of verifying an identity's access privileges

Authentication

- Using hashlib to create a hash

```
import hashlib
print("mypassword")
print("mypassword".encode())
print(hashlib.sha256("mypassword".encode()))
print(hashlib.sha256("mypassword".encode()).hexdigest())
```

```
mypassword
b'mypassword'
<sha256 HASH object @ 0x000002B38FF9A9B8>
89e01536ac207279409d4de1e5253e01f4a1769e696db0d6062ca9b8f56767c8
```

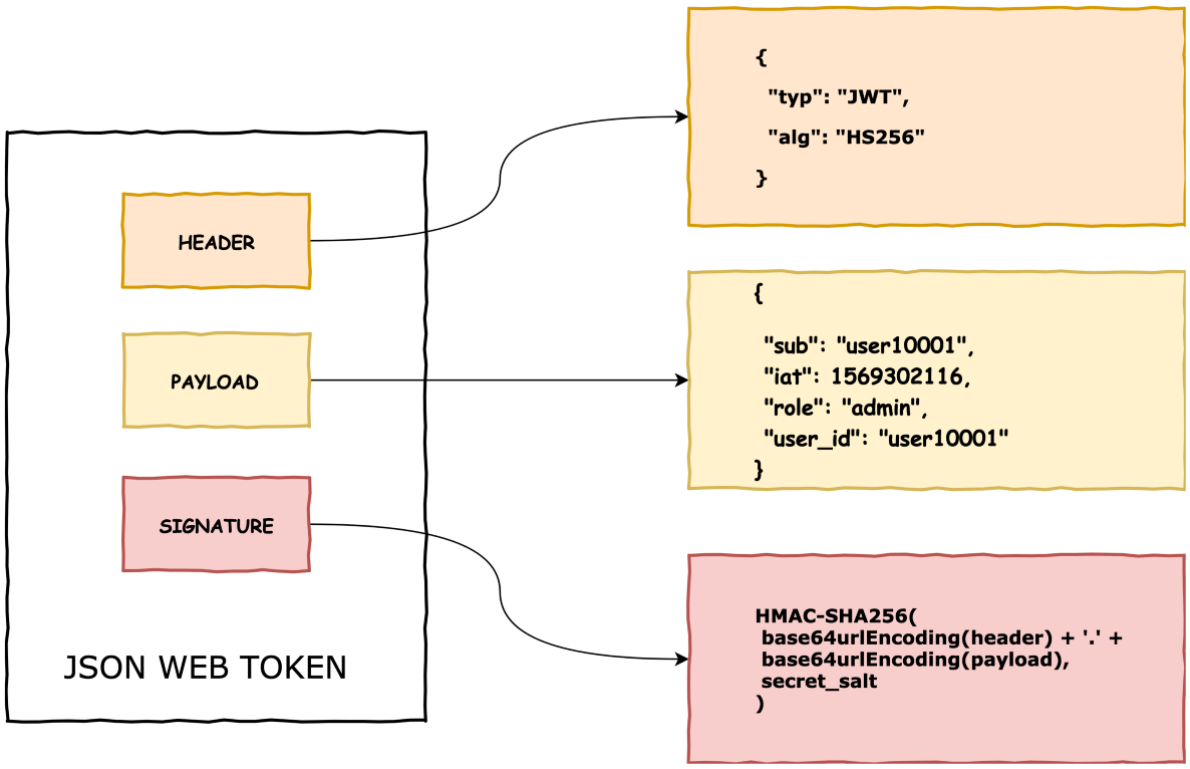
Authorisation

- 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
- 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

What is a JWT?

- 'JSON Web Tokens are an open, industry standard RFC 7518 method for representing claims securely between two parties'
- They are lightweight ways of encoding and decoding private information via a secret
- https://jwt.io/

```
import jwt
jwt.encode({'user_id': user_id, 'session_id': session_id}, SECRET, algorithm='HS256')
decoded_token = jwt.decode(token, SECRET, algorithms=['HS256'])
```



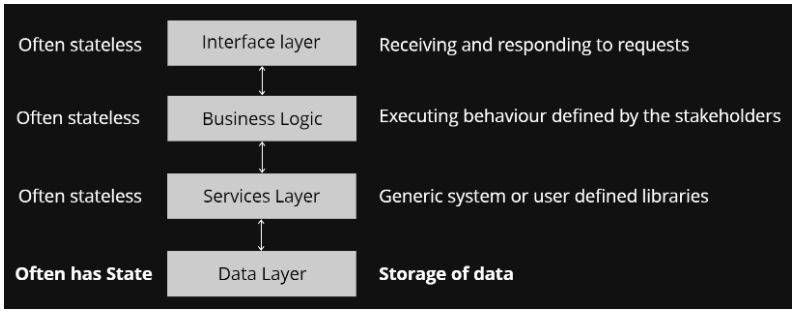
Software Engineering: 5.3 – Development – Persistence

Data

- Data: Facts that can be recorded and have implicit meaning
- From data (raw) was can find insights (information) that allow us to make decisions
- Data (and "big data") are becoming huge topics in the world of computing and mathematics

Data in Applications

- Data is part of every software (e.g. variables)
- However, often when we refer to 'Data' in software we're referring to a 'Data Layer' – a part of the software that is focused solely on housing data



Storing Data: Persistence

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

Python Specific – Pickling

- 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

```
import json
import operator
import pickle

def most_common(data):

    list = []
    count = []

    for item in data:
        if item not in list:
            list.append(item)
            count.append(1)
        else:
            idx = list.index(item)
            count[idx] += 1

    idx = count.index(max(count))
    return list[idx]

def process(data, pickle_data):
    return {
        "mostCommon": {
            "colour": f"[{data['colour']}]",
            "shape": f"[{data['shape']}]"
        },
        "rawData": pickle_data,
    }


with open('shapecolour.p', 'rb') as FILE:
    DATA = pickle.load(FILE)
    most_common_ds = most_common(DATA)

with open('processed.json', 'w') as FILE:
    json.dump(process(most_common_ds, DATA), FILE)
```

Week 7

Software Engineering: 7.1 – SDLC Requirements – Overview

- The most important part of building a system is figuring out what you need to do

SDLC (Software Development Life Cycle)

Requirements

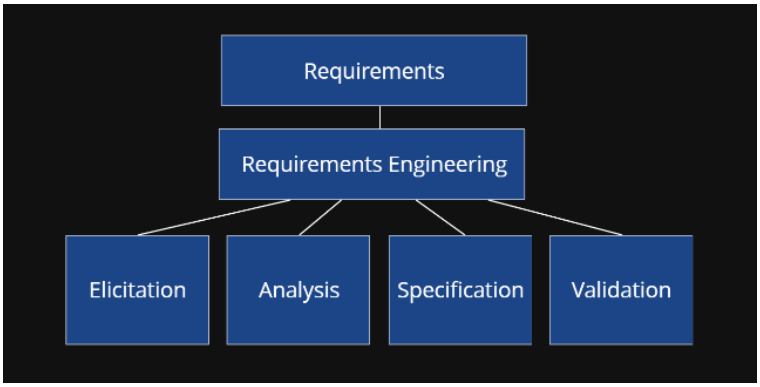
- IEEE defines a requirement as: 'A condition or capability needed by a user to solve a problem or achieve an objective'
- We would also describe requirements as:
    - Agreement of work to be completed by all stakeholders
    - Descriptions and constraints of a proposed system

Functional Vs Non-Functional Requirements

- 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

Deriving Requirements

- Requirements don't just appear in thin air. We have to derive them, and to do that we apply the process of requirements engineering



Software Engineering: 7.2 – Requirements – Use Cases, User Stories

- Requirements can be very human, and express complex flows. We need ways to model this

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

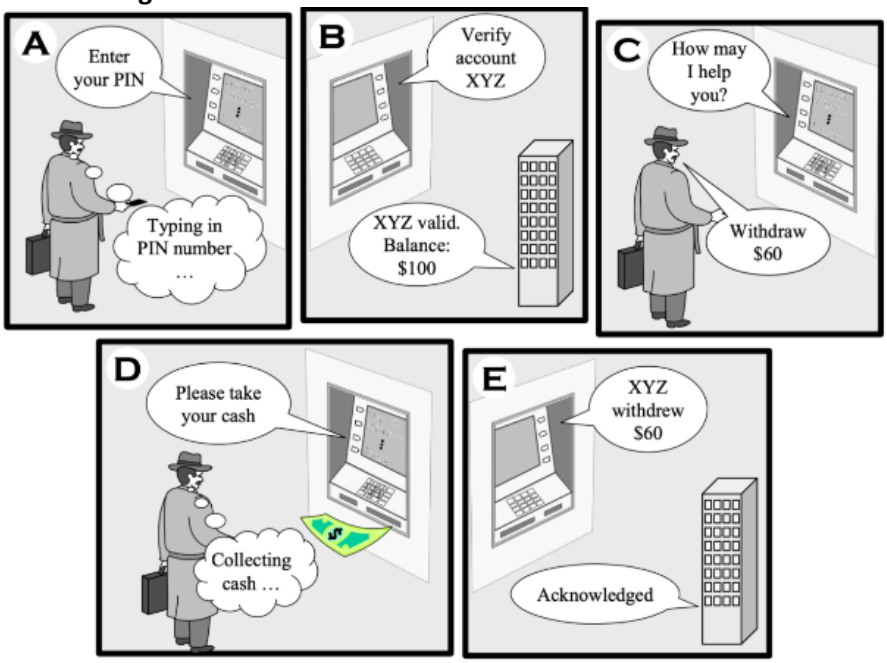
Use Cases

- 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 response with reactions
- They consider systems as a black box, and are only focused on high level understanding of flow

Use Case Representations

- Generally, you can represent use cases as:
    1. Informal list of steps (written)
    2. Diagrammatic (visual)

Use Case Diagrams



Use-case Written Form

- MAIN SUCCESS SCENARIO:
    1. ATM asks customer for pin
    2. Customer enters pin
    3. ATM asks bank to verify pin and account
    4. Bank informs ATM of validity and balance of account
    5. ATM asks customer what action they wish to take
    6. Customer asks to withdraw an amount of money
    7. ATM dispenses money to customer
    8. ATM informs bank of withdrawal

Software Engineering: 7.3 – Design – Decorators

Decorators

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

Functional Arguments

- The arguments in python can either be keywork arguments (name) or non-keyword arguments
- Non-keyword arguments cannot appear after keyword arguments in the argument list

```
1  def foo1(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      foo1('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')
```

Kwargs and Args

- We can use a generalised method of capturing:
    - *args: Non-keyword arguments as a list
    - *kwargs: Keyword arguments as a dictionary

Week 8

Software Engineering: 8.1 – Design – System Modelling

- A critical element of software design is to be able to translate complex system ideas into something high level and understandable

Conceptual Modelling

- A model is an attempt to represent a more complex system
- A conceptual model captures a system in a conceptual way
    - High level abstraction
    - Tends to be diagrammatic or visual

Conceptual models software engineers care about

- Data models
- Mathematical models
- Domain models
- Data flow models
- State transition models

How models are used

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

Communicating Models

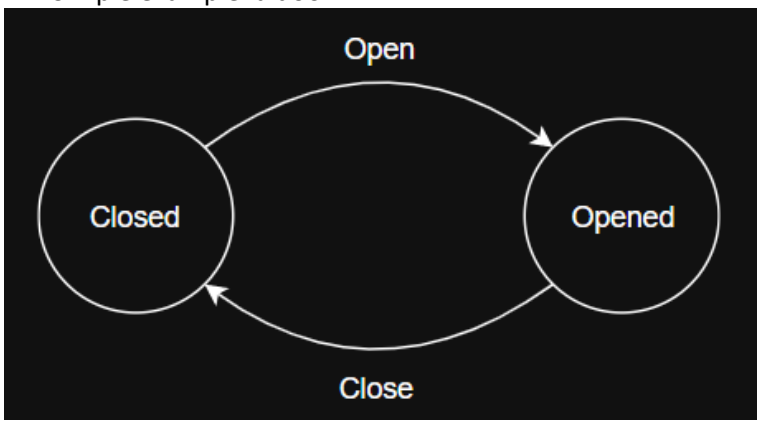
- Four fundamental objectives of communicating with a conceptual model:
    - Enhance an individual's understanding of the presentative system
    - Facilitate efficient conveyance of system details between stakeholders
    - Provide a point of reference for system designers to extract system verifications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

Conceptual Models

- Structural – Emphasise the static structure of the system
    - UML class diagrams (object orientated programming)
    - ER diagrams (database design)
    - … many others
- Behavioural – Emphasise the dynamic behaviour
    - State diagrams (state machines)
    - Use case diagram (user flows)
    - … some others

State Diagrams

- State machines made up of a finite number of states
- The machine can be transitioned from one state to another through an action
- Simple example: a door



Software Engineering: 8.2 – Deployment – Intro

- The purpose of most software is for people to use it, and for that to happen we need processes to make it available for usage

Software Deployment

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

Historical Development

- 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

Continuous Integration

- Practice of automating the integration of code changes from multiple contributors into a single software project

Continuous Delivery

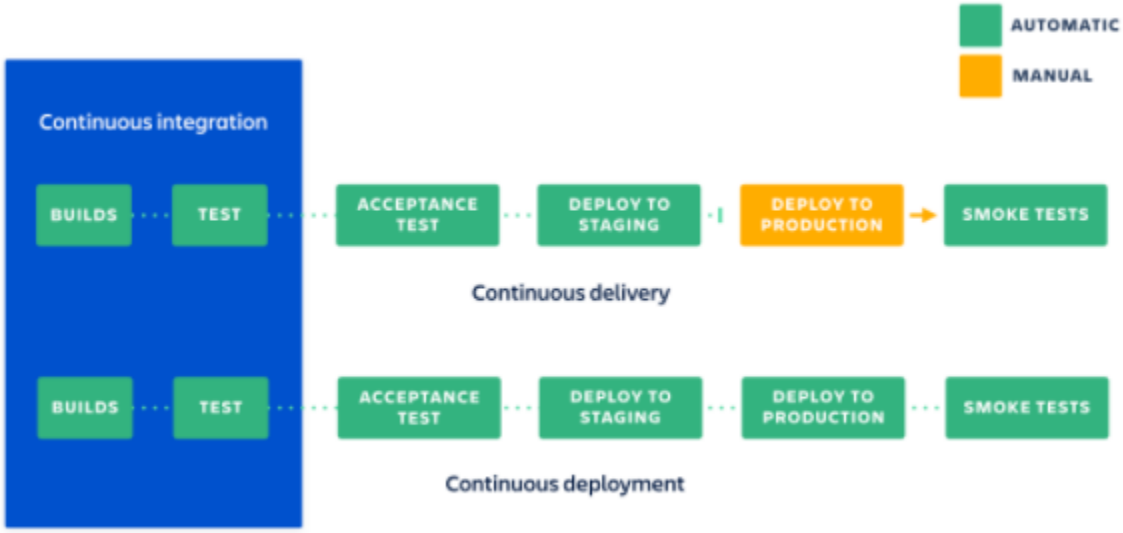
- Allows accepted code changes to be deployed to customers quickly and sustainably. This involves the automation of the release process such that releases can be done in a 'button push'

Release Methods

- Continuous delivery is often concerned with more than just going from 'your computer' to a 'production environment'
- Often, we have various stages of release, e.g. dev => test => prod
- As you move down the stages, things tend to be more stable

Continuous Deployment

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



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

Software Engineering: 8.4 – Testing – Property Based Testing

- Testing is important, but good testing can take a lot of time to write
- It is easy to miss edge cases, and this problem compounds with large code bases

Property-based Testing

- A method of testing where tests are define as general properties
- 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

Week 9

Software Engineering: 9.1 – Design – Software Complexity

- We need a constructive way to be able to understand and have conversations about how complex software is

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

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

Coupling

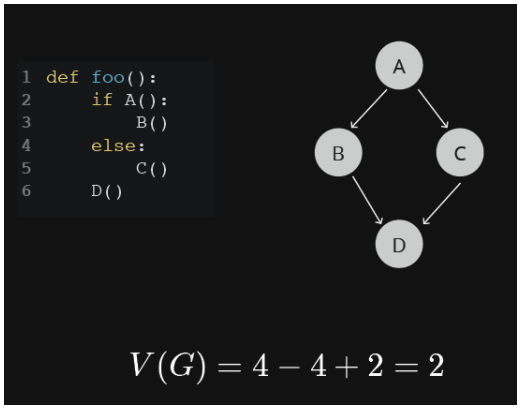
- A measure of how closely connected different software components are
- Usually expressed as a simple ordinal measure of loose or tight
- For example, web applications tend to have a frontend that is loosely coupled from the backend
- Loose coupling is good

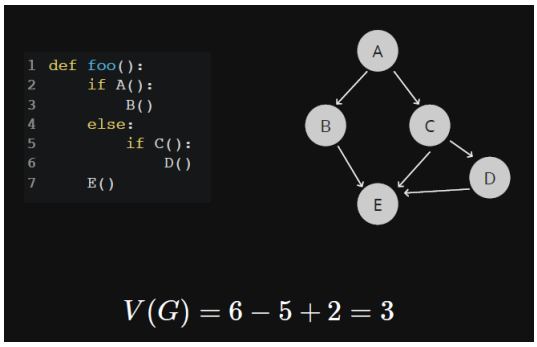
Cohesion

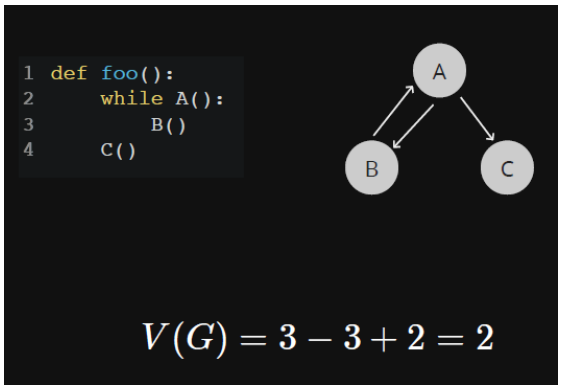
- The degree to which elements of a module belong together
- Elements belong together of a module belong together
- Elements belong together if they're somehow related
- Usually expressed as a simple ordinal measure of 'low' or 'high'
- High cohesion is good

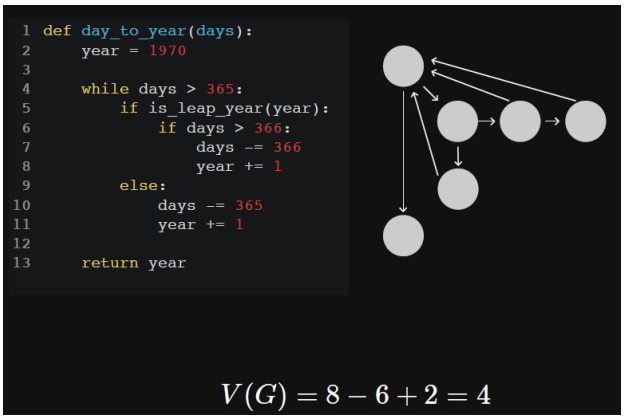
Cyclomatic Complexity

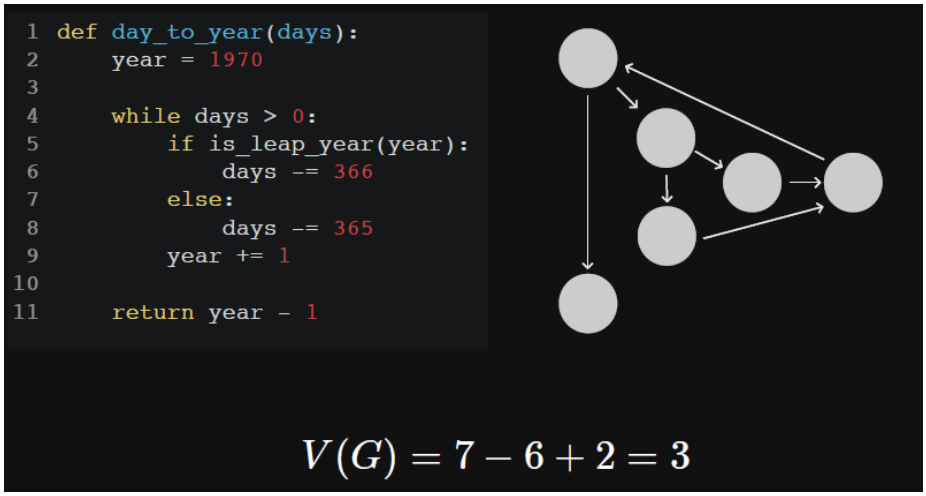
- A measure of the branching complexity of functions
- Computed by counting the number of linearly-independent paths through a function
- To compute:
  1. Convert function into a control flow graph
  2. Calculate the value of the formula $V(G) = e - n + 2$
     Where e is the number of edges and n is the number of nodes











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

Software Engineering: 9.2 – Development – Safety & Type Checking

- We need to be able to understand the characteristics of programming languages and approaches in terms of how they may prevent bugs

Safety

- Protection from accidental misuse

Security

- Protection from deliberate misuse

Software Safety

- Software becomes unsafe when its design or implementation allow for unexpected or unintended behaviours, particularly during runtime
- For example:
    - Reading uninitialized memory
    - Writing outside array bounds

Static

- Static properties can be inferred without executing the code, e.g. pylint statically checks that variables are initialised before they're used

Dynamically

- Dynamic properties are checked during execution. E.g. python dramatically checks that an index is inside the bounds of a list and throws an exception if it isn't (unlike an array in C)

Removing Errors Statistically

- 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 compile detectors a number of errors including type errors

Examples of errors

- 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

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

Memory Safety

- 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.
- C does not support exceptions at all, so errors typically have to be included in the return value

EAFP (Easier to Ask for Forgiveness than Permission)

- 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
    - Multiply different sorts of errors can be handled with one except block
- Cons:
    - Makes code non-structured
    - Harder to reason what code will be executed

LBYL (Look Before You Leap)

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

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

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 numbers

Week 9

Teamwork: 9.3 – Git – Resets & Amending

git reset --hard [hash]

- Sets all of your code to a specific commit. This is used for saying 'I want to go back in time, and I don't care about anything that's happened since that point I'm going back to'

git reset --soft [hash]

- Keeps all of your current code the same, but just changes what commit you're pointing to. This is used for saying 'I like the code that I have, so let's not change anything, but I want to alter the history of commits that got me here'

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
- The –amend flag will make the commit, but it will replace the most commit with the new commit instead of another commit to the history

git push --force
Forced push!