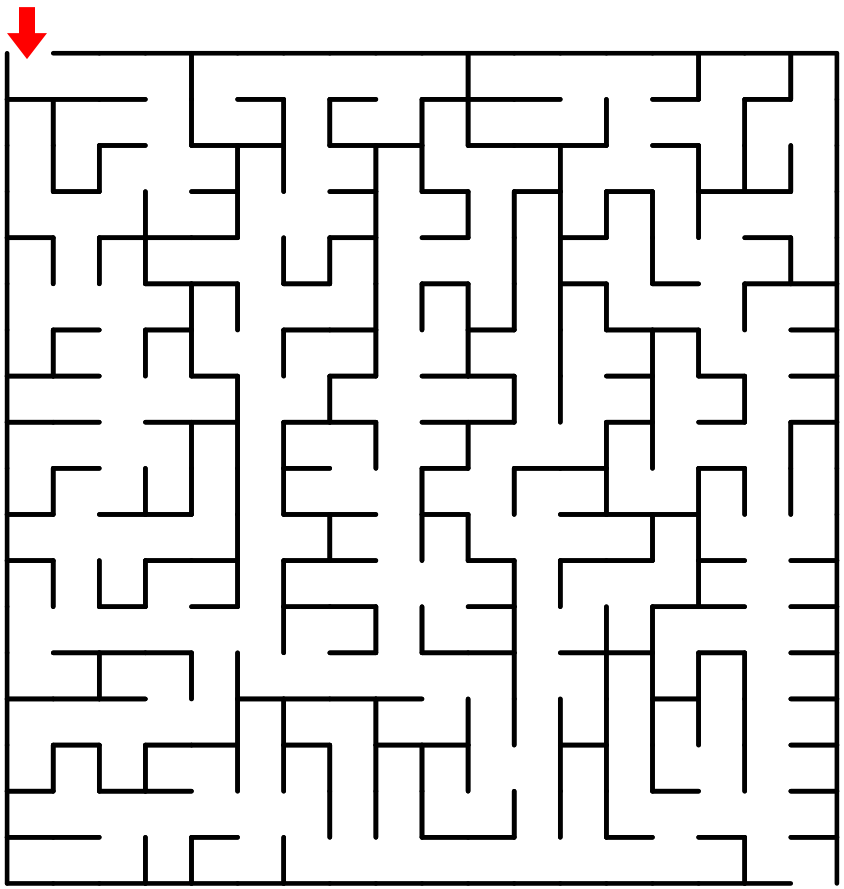
COMP1531

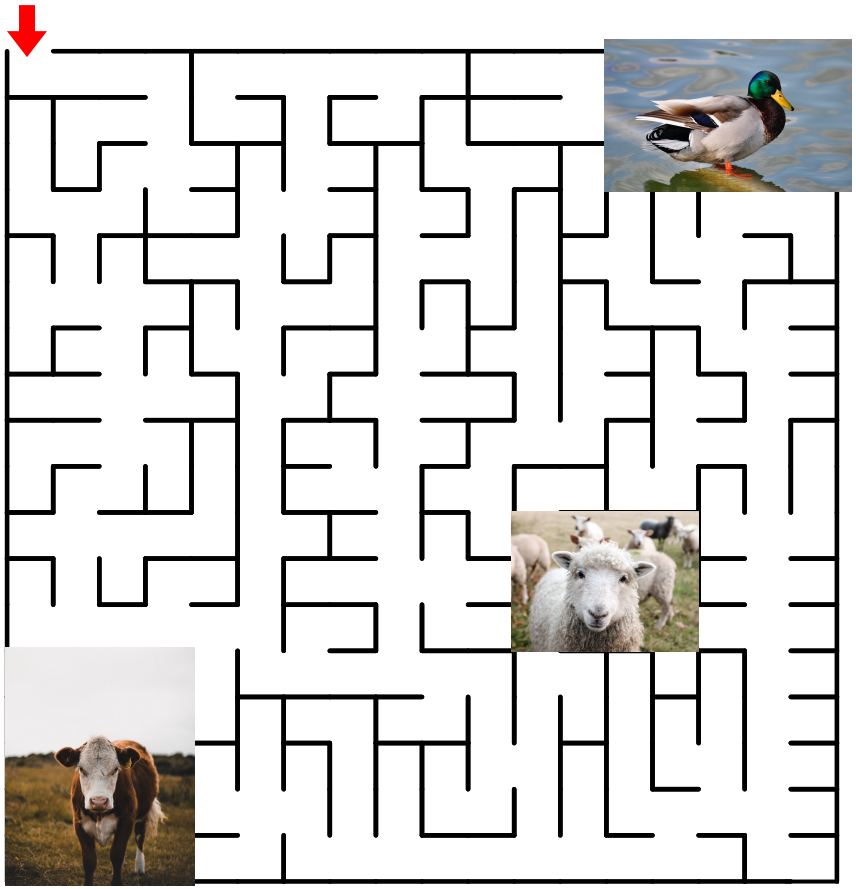
9.5 Design

WTF is design?

Programming



Design



Software design

- Understandability
- Maintainability
- Extensibility
- Reusability

- Reliability
- Performance
- Usability
- Safety
- Security
- ....

Understandability

- Can the code be understood?
- What does it mean to understand code?
- Is it sufficient to understand what it does, or do we also need to understand how it does it?

Encapsulation

- Using encapsulation, we only need to understand the externally observable behaviour of software components in order to use them
- The **other** game

Extensibility

- Being able to add new capabilities or functionality
- Making **no** changes
- Making **only minor** changes

Tic-tac-toe

- Only 3x3 with 2 players.
- What if we wanted to make it 4x4 connect 4 or even 7x6 connect 4*?
- What if we wanted to add another player?
- What if we wanted to be 3D?
- Can we make the class sufficiently extensible to allow for all these changes?
- *Should* we do that?

* See M,N,K game

Extensibility Obsession

- It's easy to focus on extensibility at the cost of other design considerations
- You'll learn more about how to make OO designs extensible in COMP2511

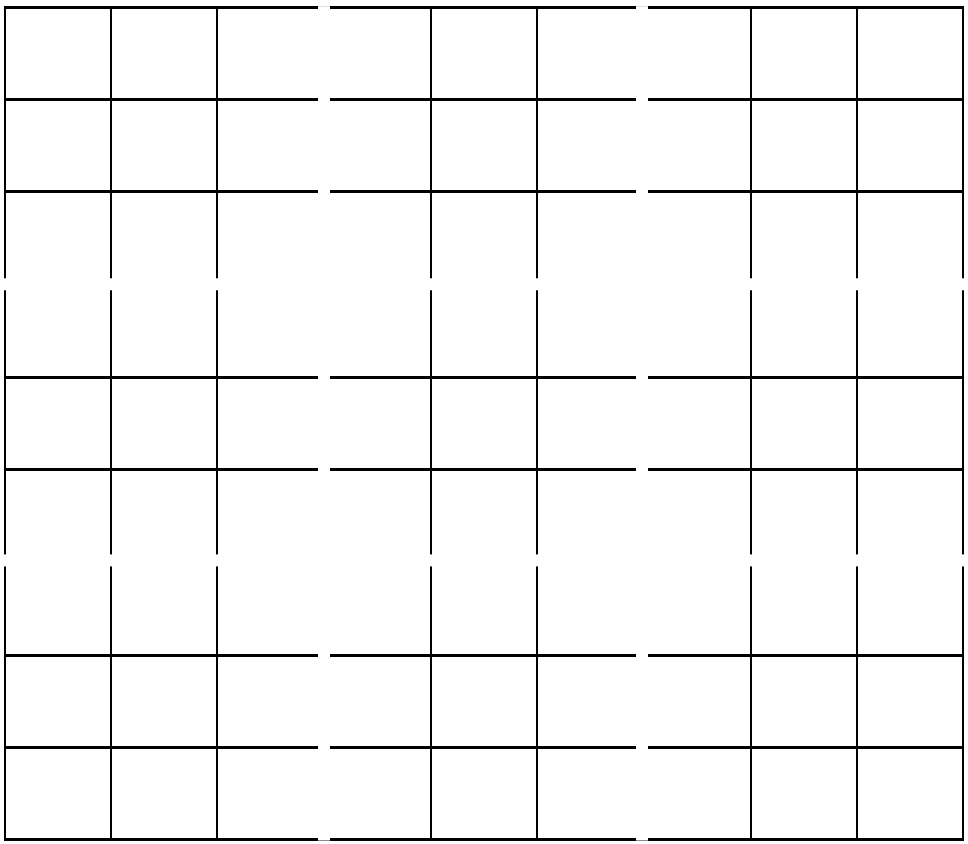
Reusability

- Can we re-use software components?
- Components that are tightly coupled to many other components are hard to re-use.
- Components with low cohesion are also hard to re-use

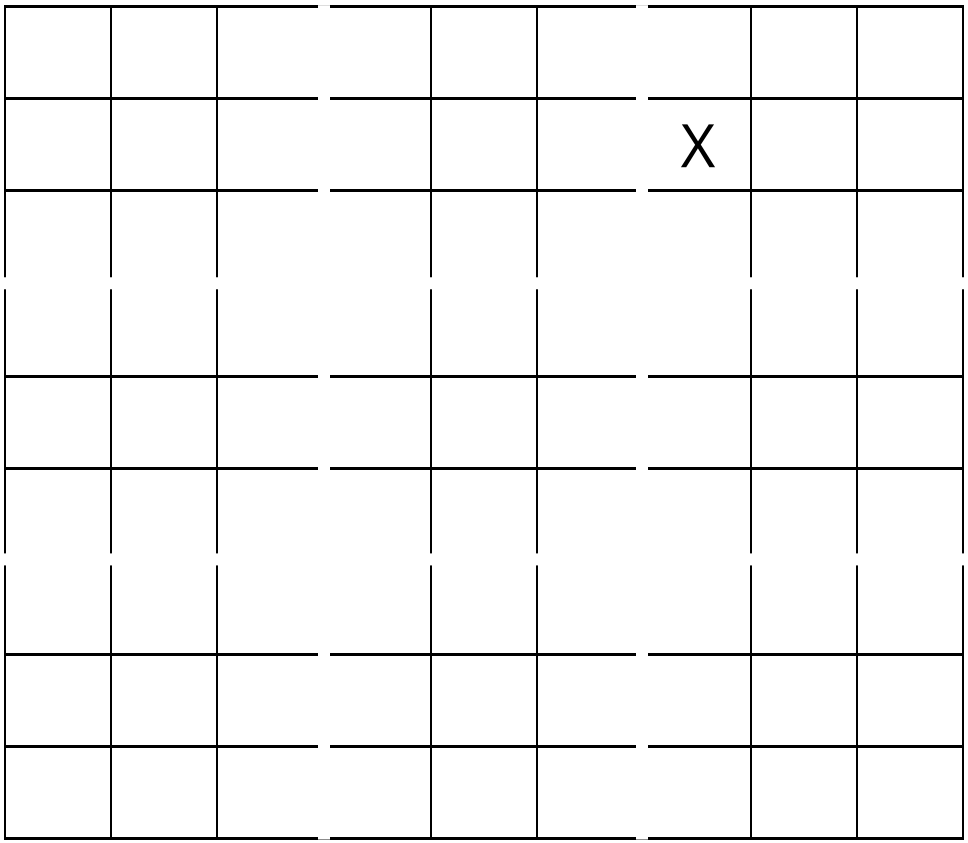
Ultimate tic-tac-toe

- A 3x3 grid of different tic-tac-toe games
- Aim: To have 3 winning games horizontally, vertically or diagonally
- Rules: The same as regular tic-tac-toe except the minor-square a player places their mark in determines the major square the other player must place their mark in.

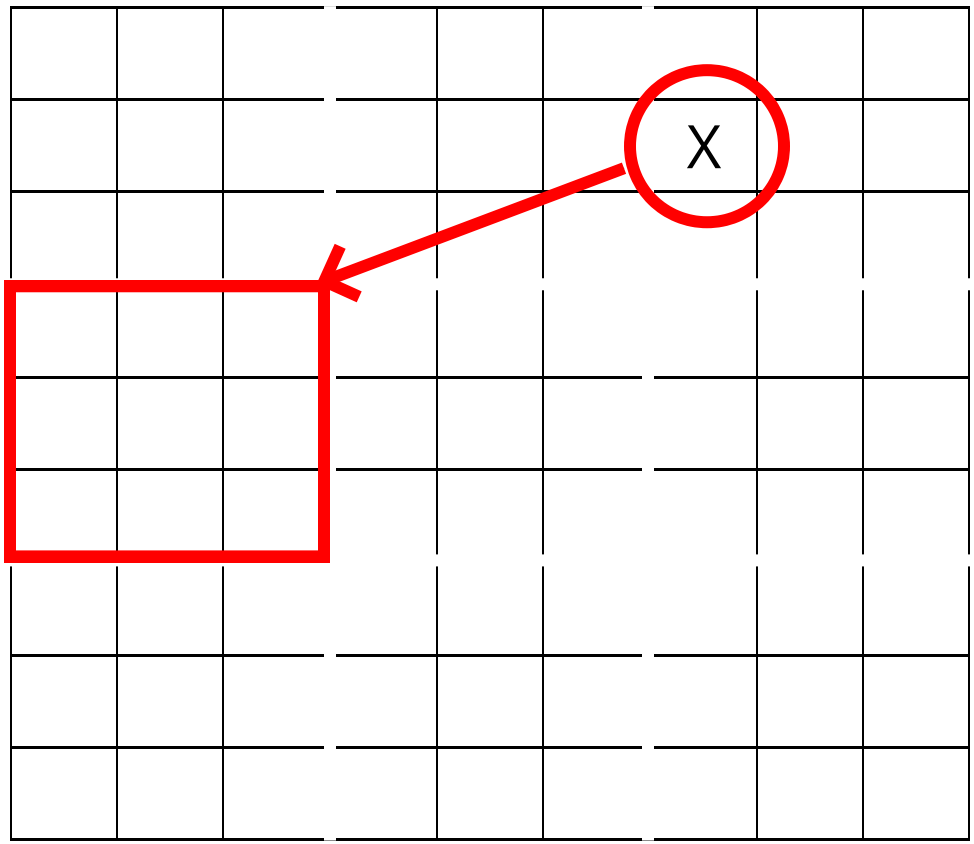
Grid is empty at the start



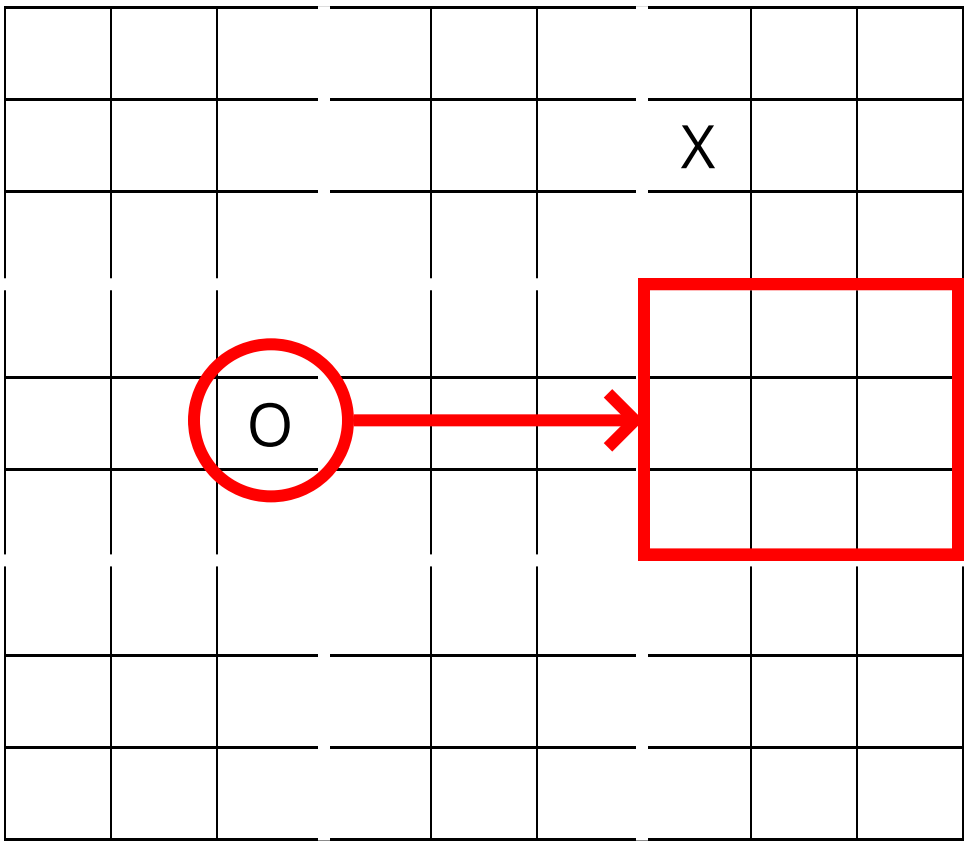
Player 1 makes their move



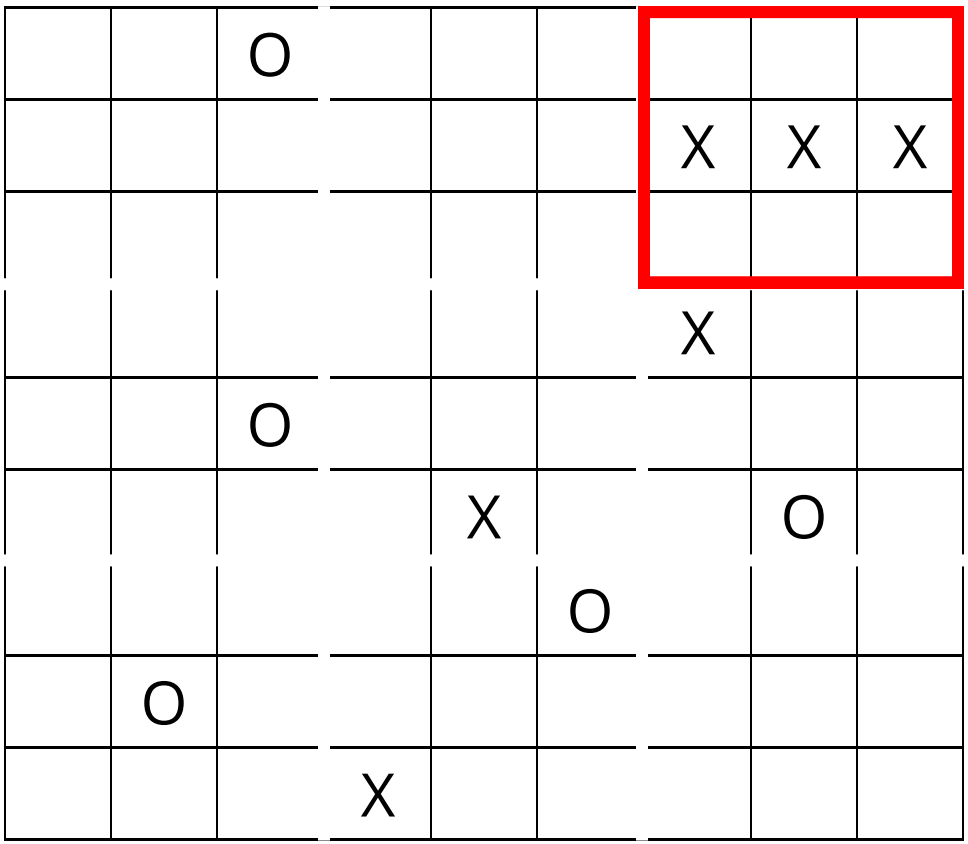
Player 2 must now make a move in the left-middle game



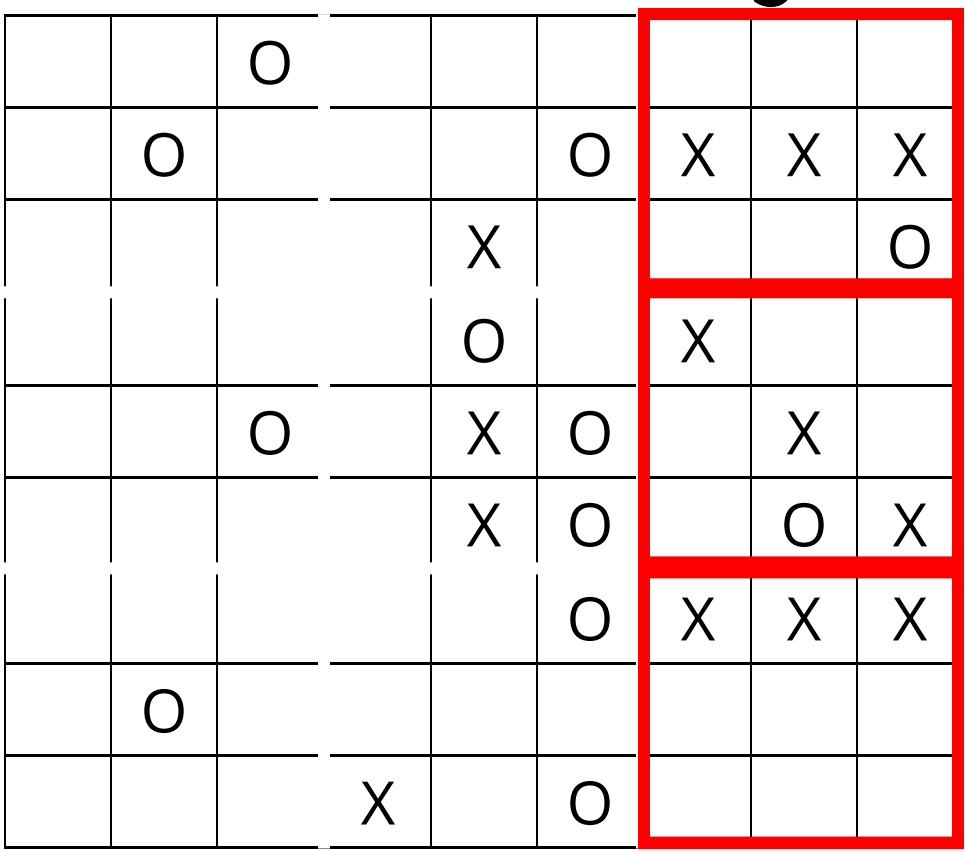
They have now "sent" player 1 to the right middle game



Player 1 has won one of the games, but that game can still be played



Player 1 has won three games vertically so wins the overall game



Ultimate tic-tac-toe

- Can we create this re-using our existing TicTacToe class?

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 }
```

Libraries

- Depending on a library can mean you rely on:
  - The author not removing the library
  - The author not breaking the library with an update
  - The author not being malicious
  - The infrastructure that provides the library being available
  - The other libraries it depends on
- Is leftpad an example of prioritising reusability over all other considerations?

Effective use of libraries

- Can we mitigate some the issues related to library usage?
- A lot of the problems are related to libraries changing without our knowledge.

Versioning

- Software products typically have version numbers or strings to differentiate their iterations
- For example, we use Python 3.7, but 3.8 is the latest version.
- On the CSE machines:

```
1 $ python3 --version
2 Python 3.7.3
```

Pinning

- Why not just pin/freeze all the dependencies to a specific version?
- Good for *applications,* because we want to build them in a reproducible way
- Doesn't work for *libraries*, because we want them to be *reusable*

Diamond Dependencies

==1.7.1

==1.6.4

Semantic Versioning

- Given a version number MAJOR.MINOR.PATCH, increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards compatible manner, and
  - PATCH version when you make backwards compatible bug fixes.

https://semver.org

Semantic Versioning

-
  - 1.7.2 - Would contain only bug fixes.
  - 1.8.0 - Would contain new features, but all previous features should work as before. Code that used the old version of the library should still work with new version.
  - 2.0.0 - May "break" previous features. This can include functions being removed, taking different arguments, or giving different output.

requirements.txt

- If we want broad (but still safe) version ranges we can define them in requirements.txt like so:

```
1 PyJWT >= 1.7, < 2.0
2 hypothesis >= 4.44, < 5.0
```

requirements.txt

These are all equivalent:

```
1 PyJWT >= 1.7, < 2.0
2 hypothesis >= 4.44, < 5.0
```

```
1 PyJWT >= 1.7, == 1.*
2 hypothesis >= 4.44, == 4.*
```

```
1 PyJWT ~= 1.7
2 hypothesis ~= 4.44
```

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
- An attempt fix to dependency hell
  - https://nixos.org/nix/