

# COMP1531

## 10.3 Safety

# Safety

Protection from accidental  
misuse

# 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

# I promised



# Static

Static properties can be inferred without executing the code

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

# 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)

# 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 most 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. E.g.

```
1 animals["fish"]
```

will thrown 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

- **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(0, 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/>