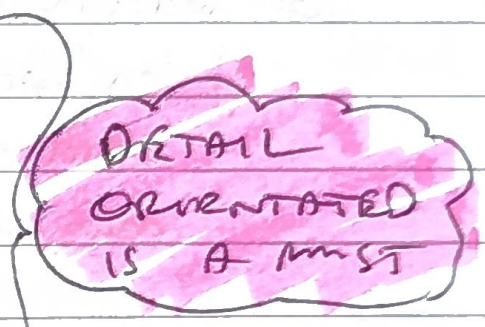
#15 - Debugging - Techniques

26/05/2025

- No strict steps to debugging
- Some useful techniques as follows:

Line-by-Line Debugging

- One of the most useful skills to develop as a programmer is to be:
  - careful
  - patient
  - read code:
    - line-by-line
    - word-by-word
    - character-by-character.



Rubber Duck Debugging

- Explaining code to a rubber-duck forces you to articulate the problem, detail by detail.
- Focus your mind in walking through the code line-by-line, capturing small detail that may lead to a deeper problem.

Debugging by Walking Away

- This technique is commonly referred to the process where your brain enters diffuse mode. Triggered by completely removing yourself from the subject.
  e.g. going for a walk/run. or even sleeping
- Important to load the problem into your brain prior to walking away.

Debugging with Print

- Implement print at strategic points in code to inspect state of data.

26/05/2025

Consider the following:

```
1  def titlize(sentence):
2      words = sentence.split()
3      new_words = []
4
5      for word in words:
6          if len(word) > 2:
7              word.capitalize()
8              new_words.append(word)
9
10     return ' '.join(new_words)
11
12 title = 'hello world of programming'
13 print(titlize(title))
```

→ Expecting `print(titlize(title))` to output:

Hello World of Programming

Instead,

Hello World programming.

Step 1

Remove any print calls i.e., print at the bottom is the only one. # print(titlize(title))

Step 2

Check that `words = sentence.split()` is working correctly

→ it should set words to a list of the words in the input string

→ include `print(words)` after line 2.

→ Running the code we get

['hello', 'world', 'of', 'programming'].

26/05/2025

This output indicates the function is correctly working up to the print invocation

Step 3

- Check whether loop is iterating over all the words in the sentence
  - → insert print(word) after line 5.
  - → the program prints the following as expected:

```
hello
world
of
programming
```

Step 4

- Check the if statement
  - → add print(word) after line 6 (if len(word) > 2)
  - → the program prints the following as expected

```
hello
world
programming
```

this tells us the if condition is working properly

Step 5

- Check the capitalization works:
  - → add print(word) after line 7 (word.capitalize())
  - → the program prints the following which is not our intention

```
hello
world
programming
```

something has likely gone wrong with word.capitalize() as we expected

```
Hello
World
Programming
```

26/05/2015

→ strings are immutable, so a method call like `word.capitalize()` won't change the value of `word`

→ Capture the return value and reassign `word` to that value.

```
line 7
    word = word.capitalize()
```

→ Now achieve the expected result:

```
Hello
World
Programming
```

→ Reinstate the final `print` invocation and we only get:

```
Hello World Programming
```

→ missing the word `of`

Step 6

→ Try to debug with `print` to get the final outcome desired.

→ I recognised that `new_words.append(word)` is with in the `if` statement and therefore only words with `len(word) > 2` are appending to `new_words`.

→ Move `new_words.append(word)` to outside the `if` statement.

27/05/2025

Inspecting with a Debugger

- `pdb` is the Python built-in debugger
  → pause program during execution
  → inspect values at run time.

NB: For Python 3.13 and later
`pdb.set_trace()` pauses program on the line it is called, rather than on the next line of code to be executed

Use `next` command to move to the next line

```
# debug.py

import pdb

counter = 1

while counter <= 5:
    print(counter)
    pdb.set_trace() # Add breakpoint
    counter += 1
```

Output:

```
1          <- This is the output of print(counter)
> /Users/...
-> counter += 1  # program execution stopped at this line
(Pdb)
```

27/05/2015

- By using the `p` command followed by the variable name, for example, `p counter`, you can access the value of any variables in scope.

```
1
> /Users/...
-> counter += 1
(Pdb) p counter  # Our command
1                # The output, the current value of 'counter'
```

- To continue execution to the next breakpoint, you can use `c` command (short for 'continue')
  → executes the code until next breakpoint

- Running `p counter` after invoking the `c` command once, should get the expected value, `2`, as `counter += 1` has been executed once.
- Use the `c` command to pause at the breakpoint for each iteration of the loop.
- To move onto the next line of the code, use the `n` command.
- To exit the debugger, type `q` and press Enter

DEBUGGING IS THE MOST VITAL SKILL REQUIRED FOR A PROGRAMMER