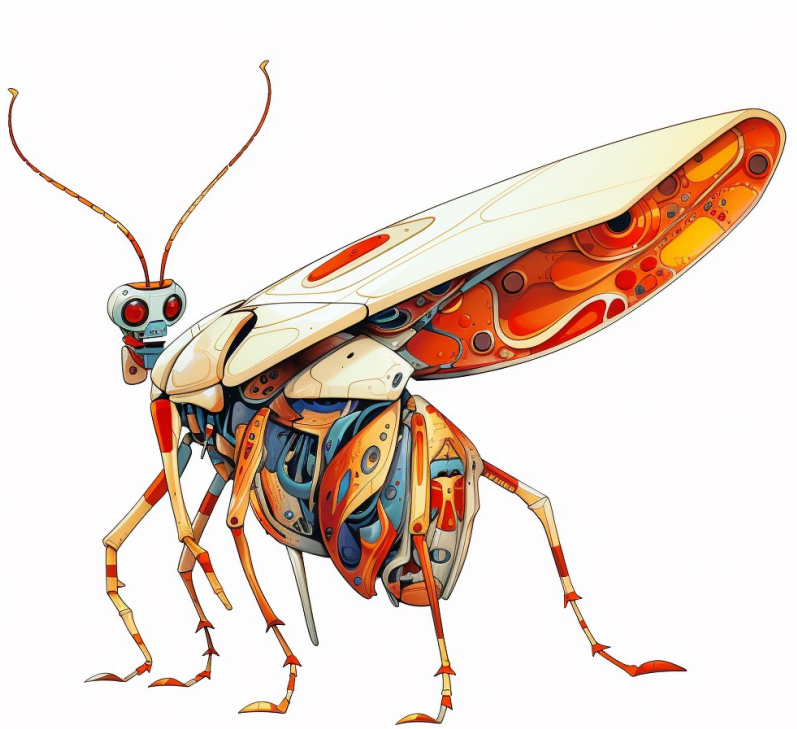
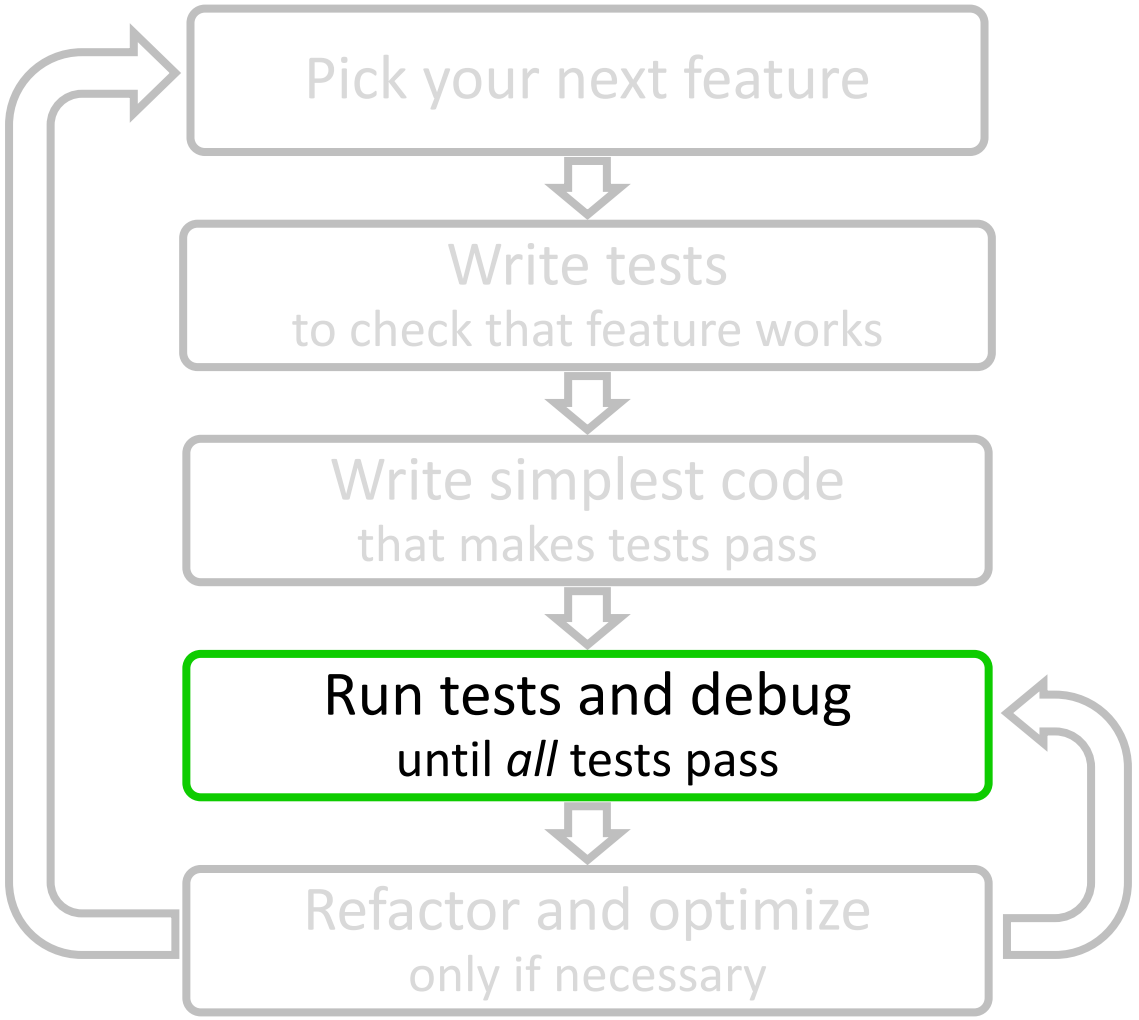
Debugging

Sometimes you can’t avoid it



Pietro Berkes and Lisa Schwetlick

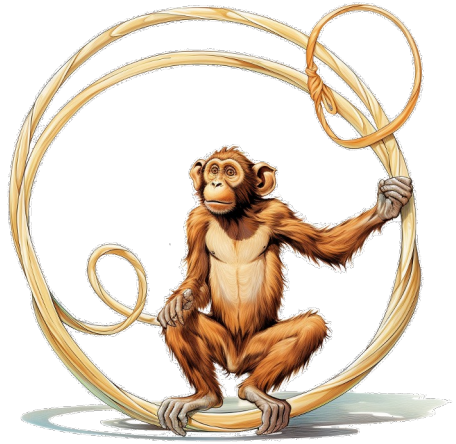
The agile development cycle



`pytest`

pdb

`timeit`
`cProfile`
`line_profiler`



Debugging

- The best way to debug is to avoid bugs
  - By writing tests, you *anticipate* the bugs
- Your test cases should already exclude a big portion of the possible causes

- Core idea in debugging: you can stop the execution of your application at the bug, look at the state of the variables, and execute the code step by step
- Avoid littering your code with *print* statements

pdb, the Python debugger

- Command-line based debugger
- pdb opens an interactive shell, in which one can interact with the code
  - examine and change value of variables
  - execute code line by line
  - set up breakpoints
  - examine calls stack

Hands-on example!

- Debugging from python
- Debugging from Jupyter
- Debugging from an IDE

Entering the debugger

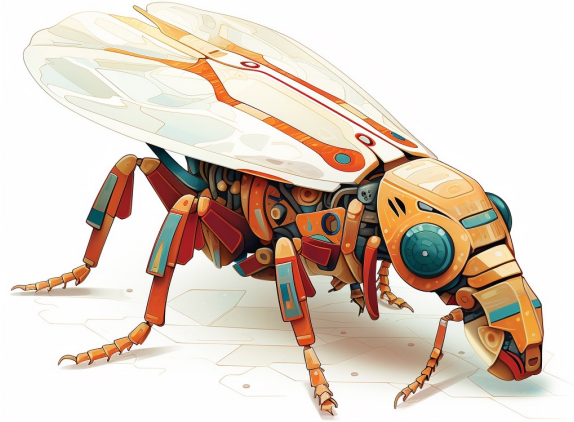
- Enter debugger at the start of a file:
  `python -m pdb myscript.py`

- Enter at a specific point in the code (easy alternative to `print`):

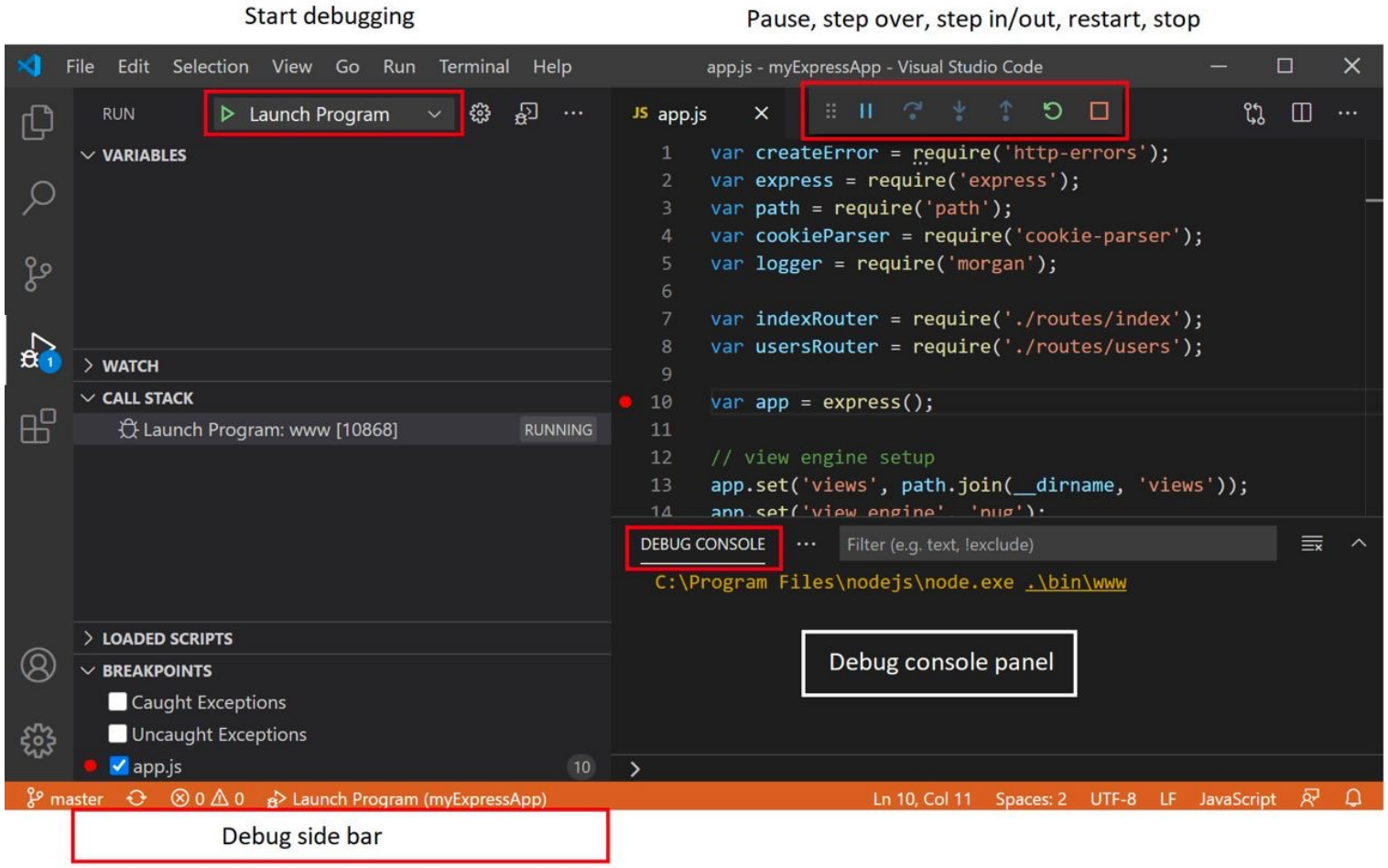
```
# some code here
# the debugger starts here
breakpoint()
# rest of the code
```

Entering the debugger from Jupyter

- %pdb – preventive
- %debug – post-mortem



Entering the debugger from VSCode



Hands-on!

- Go to `bug_hunt/file_datastore.py` and execute it

```
data = b'A test! 012'
datastore = FileDatastore(base_path='./datastore')
datastore.write('a/mydata.bin', data)

# This should pass!
assert os.path.exists('./datastore/a/mydata.bin’)
```

- It fails! But… it works when the base_path is an absolute path :-(

Hands-on!

- Fix the bug in `file_datastore.py`, using the debugger
- Submit a PR for issue #1 in the repository

pdb cheatsheet

h (help) [*command*]	print help about *command*
n (next)	execute current line of code, go to next line
c (continue)	continue executing the program until next breakpoint, exception, or end of the program
s (step into)	execute current line of code; if a function is called, follow execution inside the function
l (list)	print code around the current line
w (where)	show a trace of the function call that led to the current line
p (print)	print the value of a variable
q (quit)	leave the debugger
b (break) [*lineno* | *function*[, *condition*]]	set a breakpoint at a given line number or function, stop execution there if *condition* is fulfilled
cl (clear)	clear a breakpoint
! (execute)	execute a python command
<enter>	repeat last command

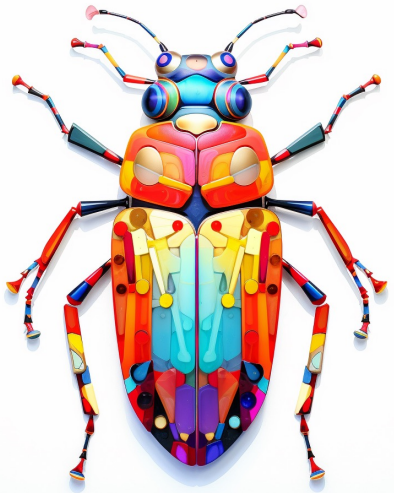
Static checking and linting

One of the problems with debugging in Python is that most bugs only appear when the code executes.

"Static checking" tools analyze the code without executing it.

- `pep8`: check that the style of the files is compatible with PEP8
- `pyflakes`: look for errors like defined but unused variables, undefined names, etc.
- `flake8`: pep8 and pyflakes in a single, handy command

- and also: `yapf, black, …`



Hands-on!

- Run `flake8` on one the files you edited today

How to react to a bug

1. Add a test that matches the behavior you expect. It will fail and reproduce the bug
2. Debug and fix the the bug
3. Run the tests until they all pass (go back to 2 if necessary)

- Now your bug is fixed *and* it will never occur again!

Up next:
Continuous Integration