Debugging

What are software bugs?

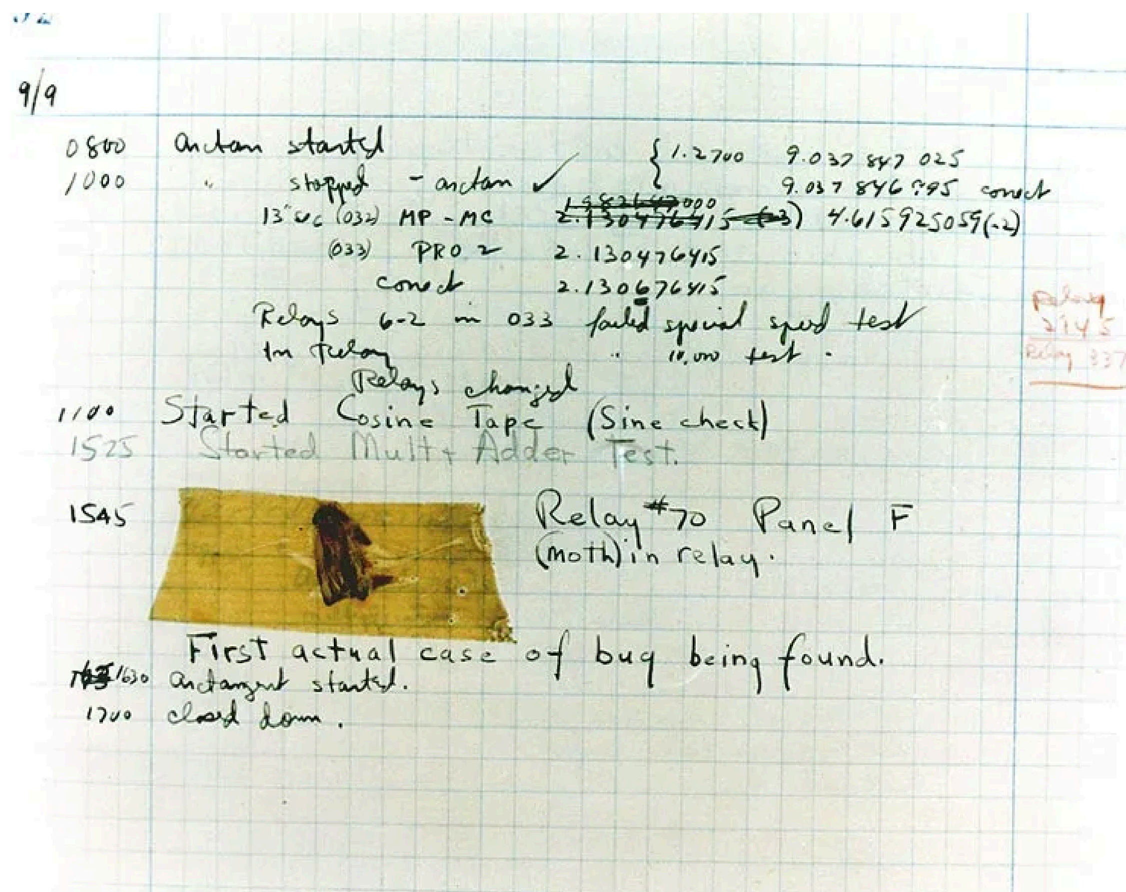
Code instructions (or the lack of) that cause:

- Program crash, freeze
- Bogus/wrong result
- Security vulnerability
- Performance degradation

CC-BY 4.0

The term “bug”

Coined in 1947 because of a literal bug jamming a relay of the Mark II Computer at Harvard University.



Pest control - Debugging!

- Any piece of software ever written had bugs.
  - Natural part of the process.
- We should minimize them and learning to debug is crutial!

CC-BY 4.0

Why not to debug

- Indicates poor testing
- Steep learning curve
- Time-consuming (expensive)
- Often mentally taxing

Nevertheless, to fix a problem one must understand it

 | CC-BY 4.0

How to debug

- Read logs / Write traces
  - Increase verbosity `level=logging.DEBUG`
- Read the code! Look where messages are printed
  - Use a rubber duck 🦆 (explain the code step-by-step)
- Be systematic
  - Narrow down the code where the error occurs
  - Bisection method is a recommended approach
- If needed use a debugger

Once completed, test for regressions!

 | CC-BY 4.0

Testing vs Debugging

Testing: increases confidence in program behaviour. Compare input, output with specification.

Debugging: finding *cause* for program error!

Testing vs Debugging

```python
def div(a, b):
  """ Returns division of `a`(dividend) by `b` (divisor) """
  result = a / b
  return result
```

Test:

```python
div(4,2) == 2
div(9,1e-323) == 9e323
```

Debug:

```python
# why does this fail?
div(9,1e-324) == 9e324
```

Hints from `Error` - best scenario

Python has many built-in Exceptions to better classify errors:

- ZeroDivisionError - `1/0`
- IndexError - `list(range(10))[11]`
- TypeError - `'3'/2`
- SyntaxError - `def = 1`

They **might** prevent bugs, and provide a hint of it's nature.

Ignoring errors, often creates **silent bugs**.

CC-BY 4.0

The Bisection Method

1. Divide code in half
2. Find which halve contains the problem
3. Repeat

Bisection Method

```python
def is_palindrome(v):
    a = list(str(v))
    b = a
    b.reverse()
    if a == b:
      return True
    else:
      return False

for value in ['ABBA', 12345678987654321, 123.321, 'abc']:
    pal = is_palindrome(value)
    print(value, "is palindrome?", pal)
```

```
ABBA is palindrome? True
12345678987654321 is palindrome? True
123.321 is palindrome? True
abc is palindrome? True
```

Example from MIT OpenCourseWare

Bisection Method (1)

```python
def is_palindrome(v):
    a = list(str(v))
    b = a
    b.reverse()
    if a == b:
      return True
    else:
      return False

value = 'abc'
pal = is_palindrome(value)
print(value, "is palindrome?", pal)
```

```
abc is palindrome? True
```

Bisection Method (2)

```
 1  def is_palindrome(v):
 2      a = list(str(v))
 3      b = a
 4      b.reverse()
 5      print(a, b)
 6      if a == b:
 7        return True
 8      else:
 9        return False
10
11  value = 'abc'
12  pal = is_palindrome(value)
13  print(value, "is palindrome?", pal)
```

```
['c', 'b', 'a'] ['c', 'b', 'a']
abc is palindrome? True
```

Bisection Method (3)

```
1  def is_palindrome(v):
2      a = list(str(v))
3      b = a
4      print(a, b)
5      b.reverse()
6      print(a, b)
7      if a == b:
8        return True
9      else:
10       return False
11
12 value = 'abc'
13 pal = is_palindrome(value)
14 print(value, "is palindrome?", pal)
```

Bisection Method (4)

```
1  def is_palindrome(v):
2      a = list(str(v))
3      b = a
4      print(a, b)
5      b.reverse()
6      print(a, b)
7      if a == b:
8        return True
9      else:
10       return False
11
12 value = 'abc'
13 pal = is_palindrome(value)
14 print(value, "is palindrome?", pal)
```

```
['a', 'b', 'c'] ['a', 'b', 'c']
['c', 'b', 'a'] ['c', 'b', 'a']
abc is palindrome? True
```

Bisection Method (5)

Mutability and aliasing.

- Variables are alias for objects
- Object may be mutable or immutable

CC-BY 4.0

Bisection Method (5.1)

```
1  a = list("X")
2  b = a
3  print("'a' same object 'b'?", a is b)
4  print("'a' is equal to 'b'?", a == b)
```

```
'a' same object 'b'? True
'a' is equal to 'b'? True
```

Bisection Method (5.2)

```python
a = list("X")
b = a.copy()
print("'a' same object 'b'?", a is b)
print("'a' is equal to 'b'?", a == b)
```

```
'a' same object 'b'? False
'a' is equal to 'b'? True
```

Bisection Method (5.3)

```
1  a = str("X")
2  b = str("X")
3  print("'a' same object 'b'?", a is b)
4  print("'a' is equal to 'b'?", a == b)
```

```
'a' same object 'b'? True
'a' is equal to 'b'? True
```

Strings are immutable!

Bisection Method (6)

```
 1  def is_palindrome(v):
 2      a = list(str(v))
 3      b = a.copy()
 4      print(a,b)
 5      b.reverse()
 6      print(a, b)
 7      if a == b:
 8        return True
 9      else:
10        return False
11
12  value = 'abc'
13  pal = is_palindrome(value)
14  print(value, "is palindrome?", pal)
```

```
['a', 'b', 'c'] ['a', 'b', 'c']
['a', 'b', 'c'] ['c', 'b', 'a']
abc is palindrome? False
```

Program execution

Don’t be intimidated, this is just a quick oversimplyfied overview

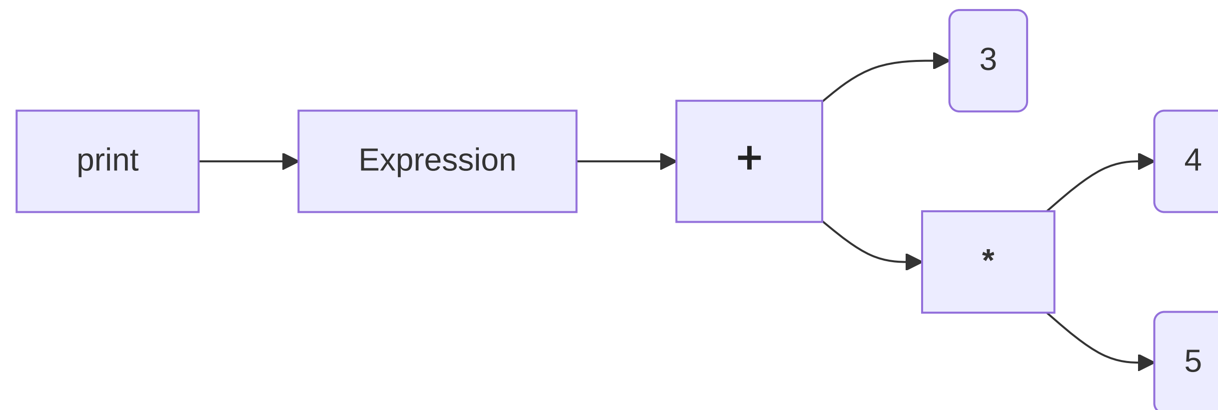
How `python` interpreter works

- Creates a tree of your code (Abstract Syntax Tree)
- Compiles each node into `bytecode` (sequence of instruction for `python` VM)
  - `bytecode` is between source code and binary
  - Sequence of *bytes* enconding **operations** and their parameters
- Executes each bytecode instruction
  - Depending on the platform, different machine code is executed

CC-BY 4.0

Abstract Syntax Tree

The AST representation for the expression `print(3+4*5)`, could be like:



Scope and Stack

- Each function call has its own **scope**.
- During execution, `python` adds and removes items from the **stack**:
  - Creates a new **frame** on function calls
  - Stores (local) variables and parameters
  - Track of <u>where</u> and <u>what</u> to return
- After the function returns, its stack frame is removed

Example

Using debuggers

Idea of debuggers:

- Investigate a program interactively
- Set breakpoints at specific lines of code
- Inspect objects and the state of the stack

Some debuggers and helpers:

- `gdb` (GNU debugger)
- `pdb` (Python debugger)
- `traceback` (Does it ring a bell?)

  Uncaught `Exceptions` show the stacktrace using `traceback`

Your IDE will likely have a debugging option you can integrate

 | CC-BY 4.0

With graphical interfaces

- Command-line debuggers can be cumbersome
- User interfaces (UI). e.g:
  - pudb - Terminal (TUI)
  - VScode - Graphical (GUI)

VScode - pdb debugger IDE integration

```
code -n main.py fibonacci.py
```

Using VSCode

Lets take a look at this implementation of the Fibonacci sequence:

main.py

```python
from fibonacci import fibonacci

if __name__ == "__main__":
    result = fibonacci(7)
    print("The 7th's Fibonacci number is: {result}")
```

fibonacci.py

```python
def fibonacci(n):
    if n == 0:
        return 0
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

- How many stack frames are created in total?
- What are the values of `a` and `b` at the end of the last iteration of the `for` loop.

CC-BY 4.0

Take home messages

- When errors messages or debug logs aren’t helpful, debug!
- Bisect method - cuts down the time spend debugging
- Debuggers
  - Great to tool follow code execution
  - Hard to master and time consuming

Shotgun Buffet

Dealing with bugs in dependencies

- Search online for the error message / problem description
- Check newer versions / releases
- Raise issue on the relevant channels (GitHub, mail, ...)
  - Describe issue, expected behaviour and obtained result
  - Make sure to provide specific information (Version, OS, ...)
  - Provide minimum reproducible example

Python Debugger

```python
1  def fibonacci(n):
2      if n == 0:
3          return 0
4      a, b = 0, 1
5      for i in range(n):
6          a, b = b, a + b
7      return a
8
9  if __name__ == "__main__":
10     result = fibonacci(7)
11     print("The 7th's Fibonacci number is:", result)
```

Python Debugger

```python
1  def fibonacci(n):
2      if n == 0:
3          return 0
4      a, b = 0, 1
5      for i in range(n):
6          a, b = b, a + b
7          breakpoint() # Python >= 3.7
8      return a
9
10 if __name__ == "__main__":
11     result = fibonacci(7)
12     print("The 7th's Fibonacci number is:", result)
```

Python Debugger(1)

```
1 python3 -m pdb fibonacci.py
```

```
> ./fibonacci.py(1)<module>()
-> def fibonacci(n):
(Pdb) h
Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where
```

Python Debugger(2)

- `b`/`clear` - create and remove breakpoints
- `l` - lists code surrounding the line being executed
- `c` - resumes the program execution (until a breakpoint is *hit*)
- `p` - prints the contents of an object
- `s` - steps into the current function call
- `bt` - prints a backtrace
- `up`/`down` - to go up or down on the call stack

A look into Python bytecode

You can compile[1]/disassemble[2] Python using python 🤯

```
1 import dis
2 source_code = "print(3+4*5)"
3 code = compile(source_code,"<string>", "exec")
4 print(f"Raw bytecode for '{source_code}': {code.co_code}")
5 print("Disassembled:")
6 dis.dis(code)
```

```
Raw bytecode for 'print(3+4*5)': b'e\x00d\x00\x83\x01\x01\x00d\x01S\x00'
Disassembled:
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 (23)
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

1. https://docs.python.org/3/library/functions.html#compile
2. https://docs.python.org/3/library/dis.html

strace[1] - Linux syscall tracer

writeloop.py

```python
import random

def write_hello_ntimes():
    with open("example.txt", "w") as file:
        number = random.randint(0,100)
        content = f"Hello {number:3}!\n"
        file.write(content)

write_hello_ntimes()
```

strace python3 writeloop.py

```
...
openat(AT_FDCWD, "/dev/urandom", O_RDONLY|O_CLOEXEC) = 4
fstat(4, {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x9), ...}) = 0
read(4, "\16\273o\312v\270T\274\36\343\236k0\17}\222\242S\351\231\264\t\26\342SD\35e\203\213f_"..., 2496) = 2496
openat(AT_FDCWD, "example.txt", O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC, 0666) = 5
fstat(5, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
ioctl(5, TCGETS, 0x7ffd5154f660)        = -1 ENOTTY (Inappropriate ioctl for device)
lseek(5, 0, SEEK_CUR)                   = 0
lseek(5, 0, SEEK_CUR)                   = 0
write(5, "Hello  74!\n", 11)            = 11
close(5)                                = 0
...
```

strace python3 writeloop.py   2>&1 | grep example.txt

```
openat(AT_FDCWD, "example.txt", O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC, 0666) = 5
```

1. https://strace.io

 | CC-BY 4.0

Further reading

Topic 20 (in ch 3) of *The pragmatic programmer* UHH Library system | MPS ebooks | German ebook via UHH