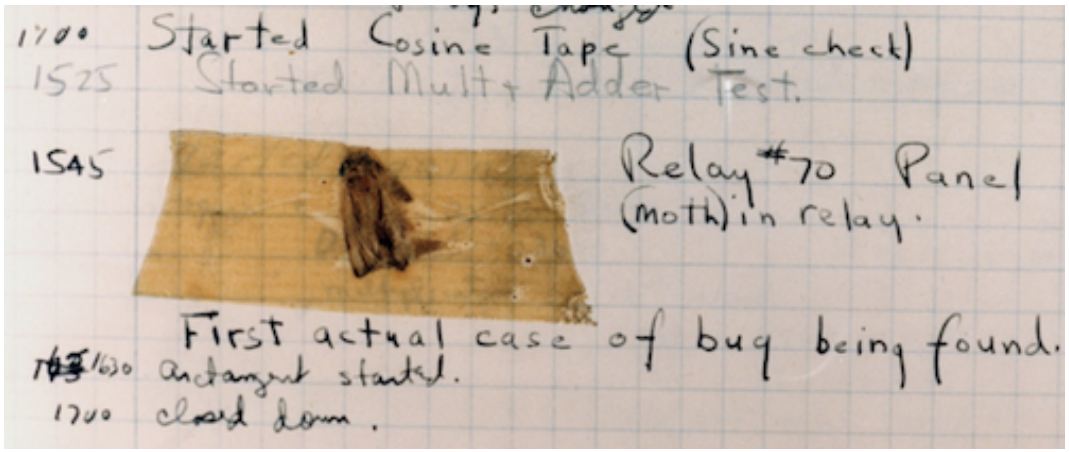
Australian National University

COMP1730/COMP6730
Programming for Scientists

Debugging and Testing

What is a “bug”?

In 1946, when [Grace] Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to **a moth trapped in a relay, coining the term bug**. This bug was carefully removed and taped to the **log book**:



Stemming from the first bug, today we call **errors or glitches** in a program a bug.

Source: `https://en.wikipedia.org/wiki/Software_bug`

The debugging process

1. Detection – realising that you have a bug, e.g., by extensive testing.
2. Isolation – narrowing down where and when it manifests.
3. Comprehension – understanding what you did wrong.
4. Correction; and
5. Prevention – making sure that by correcting the error, you do not introduce another.
6. Go back to step *1*.

Kinds of bugs/errors

1. Syntax errors
   - Easy to detect.
2. Runtime errors
   - Easy to detect (when they occur).
   - Possibly hard to understand (the cause).
3. Semantic (logic) errors
   - Difficult to detect and understand.

1. Syntax errors

* IDE/interpreter will tell you where they are.

```
  File "test.py", line 2
    if spam = 42:
            ^
SyntaxError: invalid syntax
```

```
if spam == 42:
    print("yes")

  print("spam is:", spam)
```

```
  File "../python/test.py", line 5
    print("spam is:", spam)
                          ^
IndentationError: unindent does not match any outer indentation level
```

2. Runtime errors

* Code is syntactically valid, but you're asking the python interpreter to do something impossible.
  - E.g., apply operation to values of wrong type, call a function that is not defined, etc.
  - Causes an *exception*, which interrupts the program and prints an error message.
  - Learn to read (and understand) python's error messages!

```
>>> pets = ['cat', 'dog', 'mouse']
>>> 'I have ' + len(pets) + ' pets'

TypeError: can only concatenate str (not "int") to str

>>> print(pets[3])

IndexError: list index out of range

>> print(pests[0])

NameError: name 'pests' is not defined

>>> print(pets(0))

TypeError: 'list' object is not callable
```

3. Semantic/logic errors

* The code is syntactically valid and runs without error, but *it does the wrong thing* (perhaps only sometimes).
* To detect this type of bug, you must have a good understanding of what the code is *supposed* to do.
* Logic errors are usually the hardest to detect and to correct, particularly if they only occur under certain conditions.

Isolating and understanding a fault

* Work back from where it is detected (e.g., the line number in an error message).
* Find the simplest input that triggers the error.
* Use `print` (or debugger) to see intermediate values of variables and expressions.
* Test functions used by the failing program separately to rule them out as the source of the error.
  - If the bug only occurs in certain cases, these need to be covered by the test set.

Some common errors

* python is not English.

```
if n is not int:
    ...
if n is (not int):
    ...
```

* Statement in/not in suite.

```
while i <= n:
    s = s + i**2
    i = i + 1
    return s
```

* Precision *and* range of floating point numbers.

* Loop condition not modified in loop.

```
def sum_to_n(n):
    k = 0
    total = 0
    while k <= n:
        total = total + k
    return total
```

* Off-by-one.

```
def smallest_power_of_2(n):
    """Return the smallest power of 2 that is >= n"""
    k = 1            # start at 0 or 1 ?
    p = 2
    while p <= n:    # < or <= ?
        p = p * 2
        k = k + 1
    return k         # k or k - 1 ?
```

Defensive programming

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

* Write code that is easy to read and well documented.
  - If it's hard to understand, it's harder to debug.
* Make your assumptions explicit, and *fail fast* when they are violated.

Assertions

```
assert test_expression
assert test_expression, "error message"
```

* The `assert` statement causes a runtime error if *`test_expression`* evaluates to `False`.
* Violated assumption/restriction results in an immediate error, in the place where it occurred.
* Don't use assertions for conditions that will result in a runtime error anyway (typically, type errors).

```python
# Bad practice (delayed error)

def sum_of_squares(n):
    if n < 0:
        return "error: n is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6

m = ...
k = ...
a = sum_of_squares(m)
b = sum_of_squares(m - k)
c = sum_of_squares(k)
if a - b != c:
    print(a, b, c)
```

```
# Good practice (immediate error)

def sum_of_squares(n):
    assert n >= 0, str(n) + " is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6

m = ...
k = ...
a = sum_of_squares(m)
b = sum_of_squares(m - k)
c = sum_of_squares(k)
if a - b != c:
    print(a, b, c)
```

Assertions

* Assertions testing assumptions on arguments are called preconditions.
* Assertions testing return value requirements are called postconditions.
* Other assertions should be added internal to the function testing other invariants that should hold on local variables.
* Often key invariants must hold during a correctly implemented loop- these are termed loop invariants.

Testing

Unit testing

* Different kinds of testing integration, system, user experience, etc) have different purposes.
* Testing for errors (bugs) in a component of the program – typically a function – is called *unit testing*.
  - Specify the assumptions (preconditions).
  - Identify test cases (arguments), particularly “edge cases/boundary conditions”.
  - Verify behaviour or return value in each case.
* The purpose of unit testing is to *detect bugs*.

Good test cases

* Satisfy the assumptions.
* Simple (enough that correctness of the value can be determined "by hand").
* Cover the space of inputs *and* outputs.
* Cover branches in the code.
* What are edge cases?
  - Integers: zero, positive, negative.
  - `float`: zero, very small (1e-308) or big (1e308).
  - Sequences: empty string, empty list, length one.
  - Any value that requires special treatment in the code.
  - See most_common_substr.pdf example/

pdb- the python debugger

* Using print statements to debug a script is a useful approach, but for larger scripts and more complex bugs a debugger is very useful to step through code and examine variables when an exception is thrown.
* The standard python debugger is pdb
* https://docs.python.org/3.7/library/pdb.html#debugger-commands
* This is a rudimentary command-line debugger but has the advantage of being built-in to python and so always available.
* (Your editor may have more convenient graphical interface to the debugger)

pdb- the python debugger

* The simplest approach is to add **breakpoint()** to your script at points where you need to examine the code and live variables at that point.
* (See test.py and test_debug.py code examples).

pdb- the python debugger

* Once breakpoint() is reached, the code enters the debugger, where commands can be given to inspect state etc.These include:
  - p to print a variable
  - l to list surrounding code
  - n to step to the next line
  - s to step into the next line
  - c to continue
  - u or d to step up and down stack frames
  - b to set a breakpoint

pdb- the python debugger

* We can also do post-mortem debugging (jump to the site just before an exception is thrown) without modifying the code by calling python with: **python –m pdb test.py**
* There is much more functionality available in the debugger than we cover here, such as breakpoints.
* See the documentation for more details.