Testing and Debugging

CS111 Computer Programming

Department of Computer Science
Wellesley College

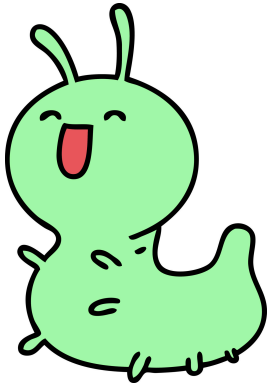
Program bugs

Bugs are mistakes in programs that cause them to behave incorrectly.

Debugging is the process of finding and fixing bugs in programs.

Testing programs reveals the presence of bugs when they don't behave as expected.

This lecture focuses on **testing** and **debugging** techniques. See the accompanying notebook for many interactive examples and exercises.



A little history

One of the most famous bugs was an actual moth discovered by Grace Hopper in a relay when she was a programmer for Harvard's Mark II Aiken Relay computer in 1947.

She had been a math professor at Vassar, was instrumental in the development of COBOL, joined the Navy in 1943, and rose to the rank Rear Admiral.

Interactive testing of Python functions in Thonny

```
# BUGGY version of countChar
def countChar(char, word):
    '''Return number of times char occurs in word, ignoring case.'''
    counter = 0
    for i in range(1, len(word)-1):
        if word[i] == char.lower():
            counter += 1
    return counter.
```

```
In [ ]: countChar('s', 'Mississippi')
Out [ ]: 4

In [ ]: countChar('p', 'Mississippi')
Out [ ]: 2

In [ ]: countChar('a', 'Mississippi')
Out [ ]: 0
```

These test results are correct.

```
In [ ]: countChar('i', 'Mississippi')
Out [ ]: 3

In [ ]: countChar('I', 'MISSISSIPPI')
Out [ ]: 0

In [ ]: countChar('m', 'Mississippi')
Out [ ]: 0
```

These test results are incorrect, indicating one or more bugs in the **countChar** function.

Towards automated testing: Printing test cases

Interactive testing is cumbersome. Can we do better?

```
def print_countChar(char, word):
    print("countChar('"      # This is just one long string
          + char + "', '"    # concatenated out of parts
          + word + "' => "
          + str(countChar(char, word)) # This is the number that
                                       # results from calling countChar
          )
print_countChar('s', 'Mississippi')
print_countChar('p', 'Mississippi')
print_countChar('a', 'Mississippi')
print_countChar('i', 'Mississippi')
print_countChar('I', 'MISSISSIPPI')
print_countChar('m', 'Mississippi')
```

```
countChar('s', 'Mississippi') => 4
countChar('S', 'Mississippi') => 4
countChar('i', 'Mississippi') => 3
countChar('I', 'MISSISSIPPI') => 0
countChar('M', 'Mississippi') => 0
countChar('m', 'Mississippi') => 0
```

Printout in Thonny.
Somewhat better than before:
No need to interactively enter test cases, but still need to inspect results

Digression: f-strings for creating complex strings

An f-string is a string preceded by the character f that specifies a string template with "holes" (marked by { }) that can be filled by the results of arbitrary Python expressions. The results of the expressions in the holes are automatically converted to strings, so we don't need to explicitly use `str` to do that.

```
def testSum(n1, n2):
    print(f'{n1} + {n2} => {n1+n2}')

In []: testSum(10,7)
10 + 7 => 17
```

f-strings make it **much** easier to create the complex strings in `print_countChar` because it's not necessary to (1) concatenate lots of strings with + or (2) convert nonstrings to strings with `str`.

```
def printCountChar(char, word):
    print(f"countChar('{char}','{word}')"
          + f" => {countChar(char, word)}"

print_countChar('s', 'Mississippi')
```

```
countChar('s', 'Mississippi') => 4
```

optimism for input/output testing

```
# file test_avg.py
import optimism as opt

def avg(a,b):
    return (a+b)//2

# create test manager for function
test_avg = opt.testFunction(avg)

# create test case and check result
case1 = test_avg.case(8,12)
case1.checkReturnValue(10)
# check will succeed

# don't need to name test
test_avg.case(7,10)\
        .checkReturnValue(8.5)
# check will fail

test_avg.case(6,4)\
        .checkReturnValue(5)
# check will succeed

test_avg.case(1,7)\
        .checkReturnValue(6)
# check will fail (bad expectation)
```

```
>>> %Run test_avg.py
✓ test_avg.py:12 ⟵ file & line number of check
✗ test_avg.py:17
  Result:
    8
  was NOT equivalent to the expected
value:
    8.5
  Called function 'avg' with arguments:
    a = 7
    b = 10
✓ test_avg.py:21
✗ test_avg.py:25
  Result:
    4
  was NOT equivalent to the expected
value:
    6
  Called function 'avg' with arguments:
    a = 1
    b = 7
```

optimism for input/output testing of countChar

```
def test_countChar():
    ''' A simple illustration of using
        optimism to perform input/ouptut
        testing on the countChar function
    '''
    tester = opt.testFunction(countChar)
    tester.case('s', 'Mississippi')\
                .checkReturnValue(4)
    tester.case('p', 'Mississippi')\
                .checkReturnValue(2)
    tester.case('a', 'Mississippi')\
                .checkReturnValue(0)
    tester.case('i', 'Mississippi')\
                .checkReturnValue(4)\
    tester.case('I', 'MISSISSIPPI')
                .checkReturnValue(4)
    tester.case('m', 'mississippi’)\
                .checkReturnValue(1)

test_countChar()
```

```
>>> %Run test_countChar.py
✓ test_countChar.py:20
✓ test_countChar.py:22
✓ test_countChar.py:24
✗ test_countChar.py:26
  Result:
    3
  was NOT equivalent to the expected value:
    4
  Called function 'countChar' with arguments:
    char = 'i'
    word = 'Mississippi'
✗ test_countChar.py:28
  Result:
    0
  was NOT equivalent to the expected value:
    4
  Called function 'countChar' with arguments:
    char = 'I'
    word = 'MISSISSIPPI'
✗ test_countChar.py:30
  Result:
    0
  was NOT equivalent to the expected value:
    1
  Called function 'countChar' with arguments:
    char = 'm'
    word = 'mississippi'
```

countChar testing with test case tuples

```
testCases = [
   ('s', 'Mississippi', 4),
   ('p', 'Mississippi', 2),
   ('a', 'Mississippi', 0),
   ('i', 'Mississippi', 4),
   ('I', 'MISSISSIPPI', 4),
   ('m', 'mississippi', 1),
]

def test_countChar2():
   ''' test countChar cases in testCases
   '''
   tester = opt.testFunction(countChar)
   for char, word, expected\
       in testCases:
     tester.case(char, word)\
       .checkReturnValue(expected)

test_countChar2()
```

```
>>> %Run test_countChar.py
✓ test_countChar.py:51
✓ test_countChar.py:51
✓ test_countChar.py:51
✗ test_countChar.py:51
  Result:
    3
  was NOT equivalent to the expected value:
    4
  Called function 'countChar' with arguments:
    char = 'i'
    word = 'Mississippi'
✗ test_countChar.py:51
  Result:
    0
  was NOT equivalent to the expected value:
    4
  Called function 'countChar' with arguments:
    char = 'I'
    word = 'MISSISSIPPI'
✗ test_countChar.py:51
  Result:
    0
  was NOT equivalent to the expected value:
    1
  Called function 'countChar' with arguments:
    char = 'm'
    word = 'mississippi'
```

Digression: Iterating over lists of tuples

```
nameTuples = [ ('Harry','Potter'),
               ('Hermione','Granger'),
               ('Ron','Weasley'),
               ('Luna','Lovegood') ]
```

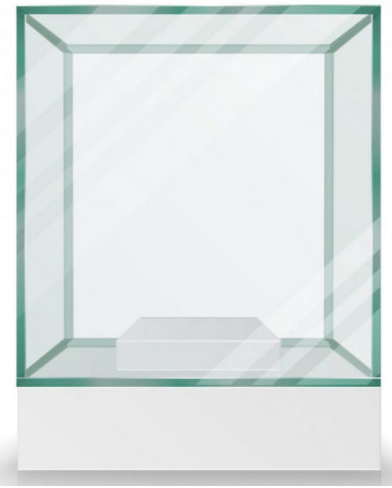
Expect each list value two be a two-element tuple. Name them `first` and `last`.

```
for first, last in nameTuples:
    print(f'First name is {first} and last name is {last}')
```

```
First name is Harry and last name is Potter
First name is Hermione and last name is Granger
First name is Ron and last name is Weasley
First name is Luna and last name is Lovegood
```

```
# The above code behaves like this
for tup in nameTuples:
    first = tup[0]
    last = tup[1]
    print(f'First name is {first} and last name is {last}')
```

Glass-box testing



Our `countChar` testing so far is an example of **glass-box testing**, which occurs when you are testing a function/program whose code you can inspect. Because you can see the implementation, you can focus on test cases that take advantage of implementation details in order to attempt to get the function to misbehave.

For example:

- You should supply test inputs that force every conditional branch in the code to be executed at least once.
- When loops are involved, you should supply inputs that cause the loop to be executed zero, one, and multiple times.
- If a loop is executed over a sequence, you should test that it processes all elements of the sequence appropriately. In particular, it should avoid errors in which it fails to appropriately process the first or last elements of the sequence.
- When sequence indices are involved, you should supply test inputs that force these indices to be edge cases.

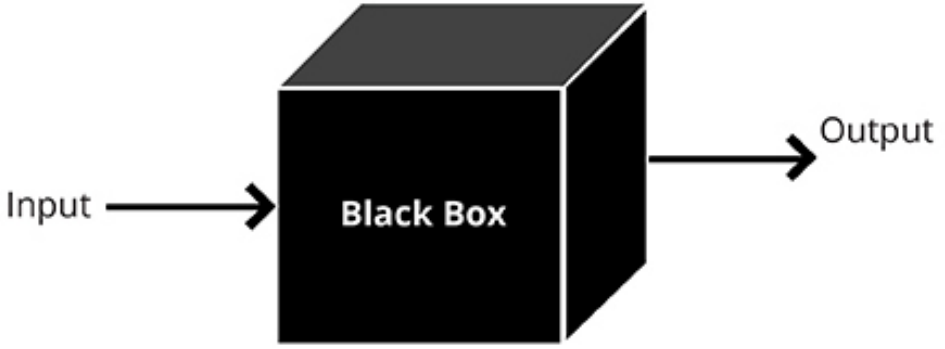
Glass-box testing and counterexamples: hasCGBlock

```
def hasCGBlock(seq):
 """ Given an RNA sequence, this function must return True if the sequence
     contains a block of 5 consecutive 'C' and/or 'G' bases, and False
     otherwise. The block may be any combination of 'C' and 'G' bases as long
     as there are 5 in a row with no other bases in between them. But if other
     bases are present, there might be more than 5 total 'C' or 'G' bases in
     the sequence without it actually containing a 'CG' block."""
    count = 0
    for base in seq:
        if base in 'CG':
            count += 1
            if count == 5:
                return True
    return False
```

The problem with this buggy **hasCGBlock** is that it just counts that the total number of Cs and Gs in word is at least 5 without checking that they are consecutive. It will behave correctly on strings with fewer than 5 Cs and Gs or with at least 5 consecutive Cs and Gs, but will incorrectly return **True** for strings that have 5 or more Cs and Gs without having 5 of them in a row.

A **minimal counterexample** is a counterexample with shortest length. Here, it's any string of length 6 with 5 Cs and Gs and one A or U that does not begin or end the string, such as `'CGAGGC'`. See the notebook for other glass-box testing examples.

Black-box testing



In **black-box testing**, the testing of the function is based purely on its input/output behavior according to its contract without being able to see the code implementing the function.

It's as if it's a mechanical contraption whose internal workings are hidden inside a black box and cannot be viewed.

Categories of black-box test cases

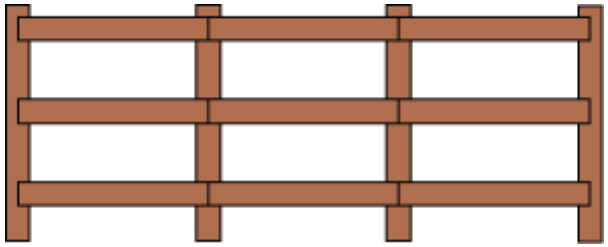
When designing black-box tests case, you must imagine ways in which the function **might** be implemented and how such implementations could go wrong. Some classes of test cases:

1. **Regular cases**: These are "normal" cases that check basic advertised input/output functionality, like tests of counting different letters in `"Mississippi"` for `countChar`.

2. **Edge cases:** These are tests of extreme or special cases that the function might not handle properly.

3. **Implied conditional cases:** When a function is supposed to take in different categories of inputs (e.g., positive or negative numbers, vowels vs. nonvowels), it implies that these categories will be checked by conditionals in the function body. Since those conditionals could be wrong, testing all combinations values from input categories is prudent.

Edge case examples



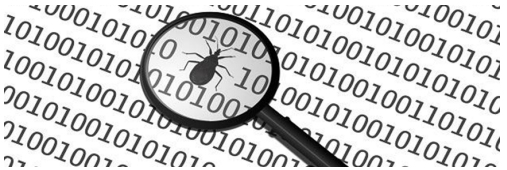
- For numeric inputs, extreme inputs can include 0, large numbers, negative numbers, and floats vs. ints.
- **Fencepost errors** are off-by-one errors, which are common in programs. E.g *n* elements in a list are separated by *n-1* commas, not *n*.
- For inputs that are indices of sequences, test indices near the ends of the sequence, e.g., indices like `0`, `1`, `-1` and `len(seq)`, `len(seq)-1`, `len(seq)+1`. Since Python allows negative indices, you should also test `-len(seq)`, `-len(seq)-1`, `-len(seq)+1`.
- For functions involving elements of sequences, test elements in the first and last positions of the sequences, e.g. characters at the beginning and end of a string.
- For inputs that are sequences, empty and small sequences are often not handled correctly, so you should always test empty and singleton strings/lists/tuples. When specific numbers are mentioned in the contract (e.g. `isCGBlock` tests for 5 consecutive values) it's important to test strings of length <= 5 as edge cases.
- For inputs expected to be booleans, what happens if other Truthy/Falsey values are supplied? Is it OK to treat other Truthy/Falsey values as True/False?

Black-box test cases for `countChar`

When testing functions like countChar with string inputs, we don't need real English words or long strings. A variety of shorter strings can suffice.

```
blackBoxCountCharTestCases = [
  ('a','', 0), # Test the empty string
  ('a','b', 0), # Test "negative case" singleton string
  # Test all capitalizations of "positive" singleton string
  ('a','a', 1), ('a','A', 1), ('A','a', 1), ('A','A', 1),
  # Test 2-element strings (char can be at beginning/end of word)
  ('a','Aa', 2), ('a','aA', 2), ('A','Aa', 2), ('A','aA', 2),
  # No need to repeat capitalization combinations here:
  ('a','ab', 1), ('a','ba', 1), ('a','bb', 0),
  # Length-3 strings distinguish ends from middles
  ('a', 'aaA', 3), ('a', 'aAA', 3), ('A', 'aaA', 3), ('A', 'aAA', 3),
  ('a', 'aab', 2), ('a', 'aba', 2), ('A', 'baa', 2), ('A', 'aAA', 2),
  ('a','abb', 1), ('a', 'bab', 1), ('a','bba', 1), ('a', 'bbb', 0),
  # Try a few longer strings
  ('a','aAAaA', 5), ('A','aAAaA', 5),
  ('a','abAbA', 3), ('A','abAbA', 3),
  ('a','babAb', 2), ('A','babAb', 2),
  ('a','bbbbb', 0),
]
```

Debugging Techniques



Test cases help us determine **cases in which** functions misbehave. But then how do we determine **why** they misbehave and **how** do fix them?

The notebook contains examples of these **debugging techniques** for identifying and fixing bugs in programs:

1. Pay Attention to Error Messages
2. Use `print` to show a function call with its arguments
3. Use `print` to show the return value of a function
4. Use `print` to show both calling and returning from a function
5. Using `print` to display iteration tables

Many programming environments (including more advanced versions of Thonny) provide additional debugging tools that allow stepping through programs line-by-line, examining the values of variables, navigating data structures, etc.

Peter Mawhorter's Debugging Handout
(available from the Reference tab on the CS111 web site)

1 Probe

Add `print` statements. Use to:

- Check if a function is being called or not:

```
def f(x, y):
    return x + 3*y
```
→
```
def f(x, y):
    print("HELLO FROM f")
    return x + 3*y
```

- Check the value of a variable:

```
y = 15 / x
```
→
```
print("x:", x)
y = 15 / x
```

- Check what happens at a conditional:

```
if x > 5:
    y = 10
else:
    y = 3
```
→
```
if x > 5:
    print("x > 5")
    y = 10
else:
    print("x <= 5")
    y = 3
```

2 Trace

Use multiple **probes** to understand code. Use to:

- Figure out where a value comes from:

```
def f(a):
    g(a * 3)

def g(b):
    for i in range(b):
        h(9-i)

def h(c):
    print(10/c)
```
→
```
def f(a):
    print("a:", a)
    g(a * 3)

def g(b):
    print("b:", b)
    for i in range(b):
        print("i:", i)
        h(9-i)

def h(c):
    print("c:", c)
    print(10/c)
```

(error if `c` is 0 in function `h`)

3 Unpack

Split up a complicated expression into multiple statements. Use this to:

- Isolate an error in a complex expression:

```
x = function(
  (a + 3*b)/(c * d),
  b / a
)
```
→
```
top = a + 3*b
bot = c * d
fst = top / bot
sec = b / a
x = function(fst, sec)
```

(ZeroDivisionError on line 1)

(ZeroDivisionError on line 4, so `a` must be the problem)

4 Toggle

Turn a line of code into a comment. Use to:

- Disable (can later re-enable) optional code:

```
def f(a, b):
    print("R: ", a/b)
    return a + b + a
```
↔
```
def f(a, b):
    #print("R: ", a/b)
    return a + b + a
```

- Temporarily replace broken code with a dummy value:

```
x = (3*y + 4*z)/w
```
→
```
#x = (3*y + 4*z)/w
x = 9
```

5 Bisect

Comment out half of your code to find the half that works, and then half of the broken part, etc., until you isolate an error. Use this to:

- Find missing brackets or commas:

```
pairs = [
  [0, 1],
  [10, 11,
  [20, 21],
  [30, 31],
]
```
→
```
pairs = [
#  [0, 1],
#  [10, 11,
  [20, 21],
  [30, 31],
]
```

(syntax error at end of file)

(works now, so error must be in the commented zone)

Note: To fit examples on this page, short and meaningless variable names have been used. ***DO NOT*** *do this in your own code.*