Topic 7
Testing and Exceptions

CS 1MD3 • Introduction to Programming
Winter 2018

Dr. Douglas Stebila

McMaster University

TESTING

A program with a bug

Write a function max_product that takes as input a base 10 number s (represented as a string), and outputs the product of the **twelve**

```
def max_product(s):

    MAX_PRODUCT = 0

    for i in range(len(s)-12):

        prod = 1
        for j in range(12): prod *= int(s[i+j])
        if prod > MAX_PRODUCT:
            MAX_PRODUCT = prod
```

What is the bug?

How could I have detected the bug?

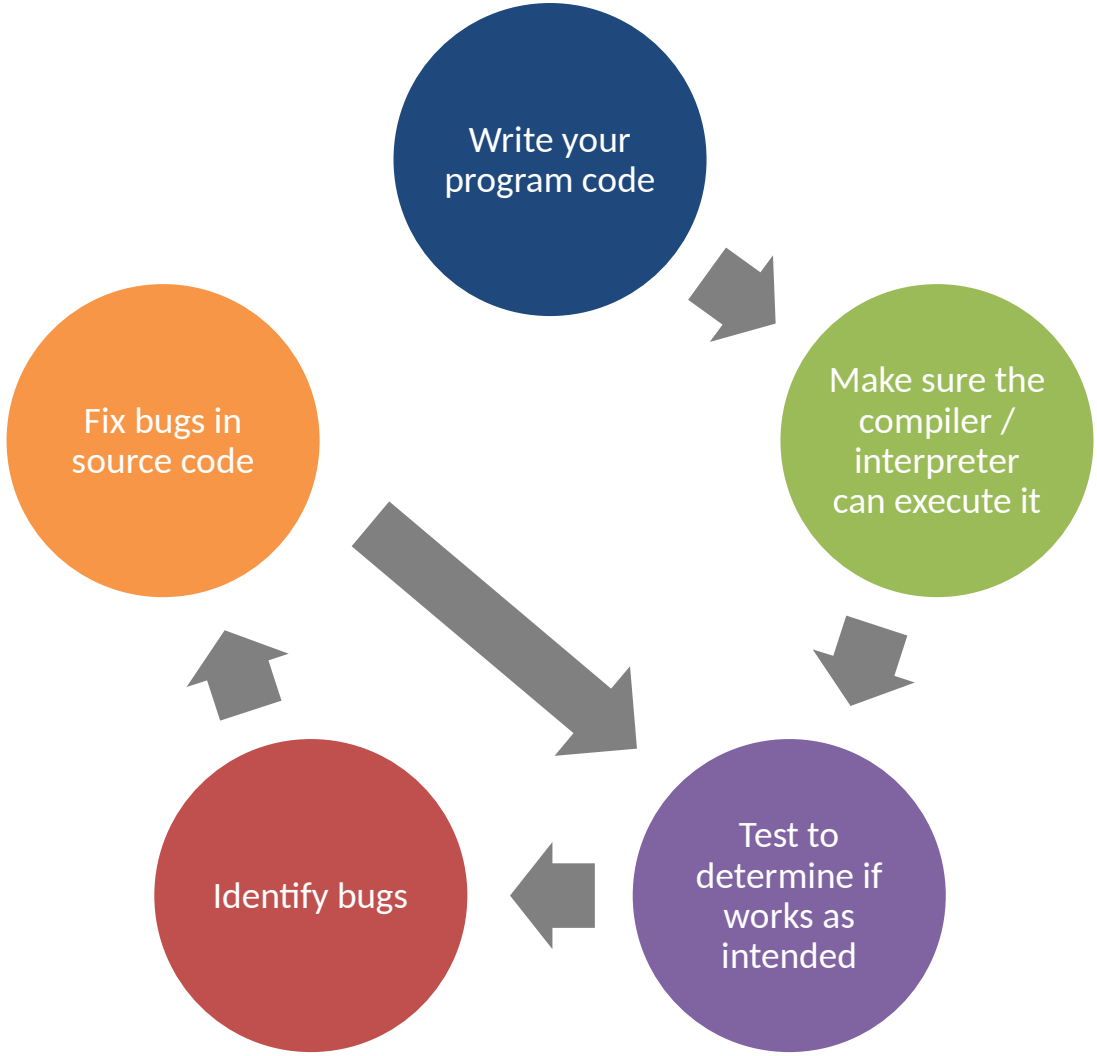
- Make some test cases with known correct answers to see if I get the right answer.
- Have another person independently implement the function and see if we get the same answer.
-

Testing and debugging

- The goal of **testing** is to determine whether a program works as intended or not

- **Debugging** is the process of trying to fix a program that you already know has a bug

Testing and debugging cycle



Testing methods

White-box techniques

- Rely on knowing the program's source code

Black-box techniques

- Don't rely on knowing the program's source code
- Only rely on the ability to put an input in and see the corresponding output

Both require a clear **specification** of the program's expected behaviour.

Black-box techniques

- Don't rely on knowing the program's source code
- Only rely on the ability to put an input in and see the corresponding output
- Focuses on the development of **test cases** to see if the actual output matches the expected output for a given input

Software design principles

- **Modularity / decomposition**: a program is broken into parts (functions) that are
  - reasonably self-contained
  - achieve a clear purpose, and
  - can be reused
- **Abstraction:** a component (function) of a program can be used without knowing how it achieves its goal

Black-box testing of big programs

- **Modularity** enables better black-box testing
- Develop test cases for every function within a program to ensure each individual function works as expected
  - This is called **unit testing**
- Then develop tests to determine if the program as a whole works as expected
  - This is called **integration testing**

Regression testing

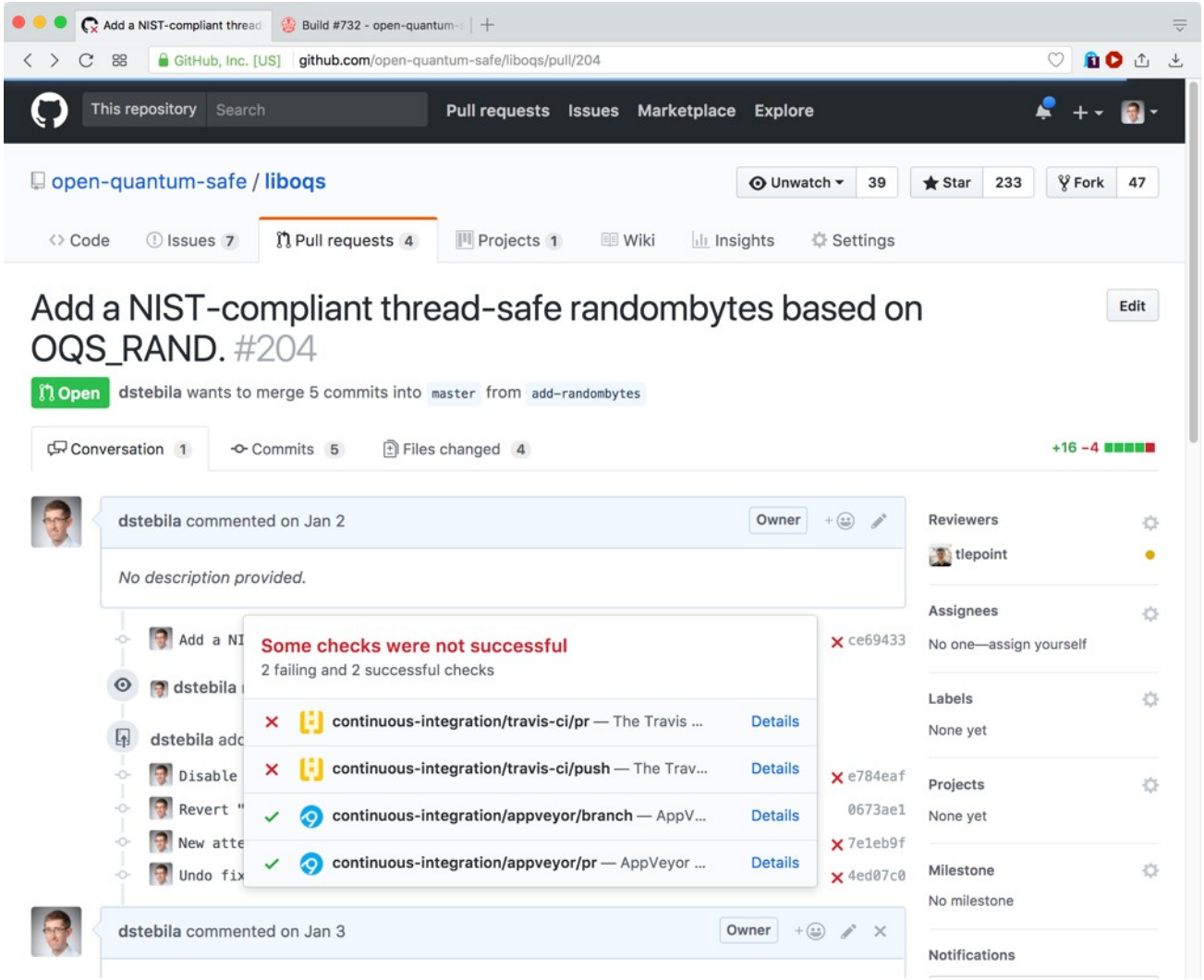
- **Regression**: when a bug is fixed, and then later more source code changes are made that reintroduce the same (or a similar) bug

- When fixing a bug, develop two test cases:
  1. One that should fail if the bug is present
  2. One that should pass once the bug is fixed
- Keep both test cases around after the bug is fixed
-

Continuous integration testing

- Create automated testing framework that runs all your tests every time you make a change to the code
  - Or at least every time you submit your changes to your organization's source code repository

Continuous integration testing example



Developing test cases

- Want to cover all different types of input that could be provided
- Need to take into account **edge cases:** when the input is at some extreme or might require special handling
  - For numbers: positive, negative, 0, 1, odd, even, prime, composite, …
  - For strings: empty string, 1 character string, …
  - For lists: empty list, 1 element list, …
  - For subsequences: at the beginning, at the end

Edge cases for max_product

- Input string is minimum length (12 digits)
- Max product is minimal: 0
- Max product is maximal: 9*9*9*…*9
- Substring corresponding to max product starts at the beginning of the string
- Substring corresponding to max product starts at the end of the string

Sample test case for max_product

- max_product('12345678912345678 9') == 121927680

- Is an edge case test: substring corresponding to max product starts at the end of the string
- But not actually correct: max product is 7*8*9*1*2*3*4*5*6*7*8*9 =

Testing my test case

- I should have tested the test case using an independent method I knew to be correct
  - For example, a calculator

- But not always possible to have an independent method
- If not, try to design test cases that check consistency
- For example, adding a 1 at the end of the string shouldn't change the max product:

```
max_product('123456789123456789 1') == max_product
```

A program with a bug

Write a function max_product that takes as input a base 10 number s (represented as a string), and outputs the product of the **twelve** adjacent digits that have the greatest product.

```
def max_product(s):

    MAX_PRODUCT = 0

    for i in range(len(s)-12):

        prod = 1
        for j in range(12): prod *= int(s[i+j])
        if prod > MAX_PRODUCT:
            MAX_PRODUCT = prod

    return MAX_PRODUCT
```

Should be 11

An "off-by-one" bug.

The effect is that it never included the last character of the string in the loop

Question 8 of Midterm Test 1

- The sample test cases for questions 8a and 8b were wrong
- Questions 8a and 8b worth 3 marks out of 30
- We will mark the test of 25
  - Marks above 25 are bonus that will be applied to final grade
- We will still mark question 8a and 8b
  - Full marks if you wrote an algorithm that followed the question specification and satisfies corrected test cases (even though the sample test case fails)
  - OR if you wrote an algorithm that works based on the sample test cases

White-box techniques

- Rely on knowing the program's source code

- **Code review**: A process where two or more developers visually inspect program code, typically several times
  - Check for programming best practices
  - Look for potential inefficiencies
  - Look for common vulnerabilities
- Code review tools can help the process
- There are formal code review methodologies
- Can be very time consuming

White-box techniques

- **Code coverage**: Do the test cases exercise all lines of our source?
  - E.g. If we have an if statement that handles even numbers in one branch and odd numbers in another branch, do our test cases include both even and odd numbers?
- **Static analysis**: Run automated tools to find flaws in programs.
  - E.g. Do I try to use a variable after it's been freed from memory? Is it possible for a counter to go past end of an array?
  - Less relevant for Python than other languages like C

DEBUGGING

Debugging

- Once your testing has shown that your program doesn't behave as intended, you have to identify the bug and then fix it

To find the bug:

1. Need a test case that reliably fails
2. Try to understand the internal state of the algorithm

Debugging

- To understand the internal state of the program:
  - Use print statements to print out potentially relevant variables.
  - Use a debugger to step through the program

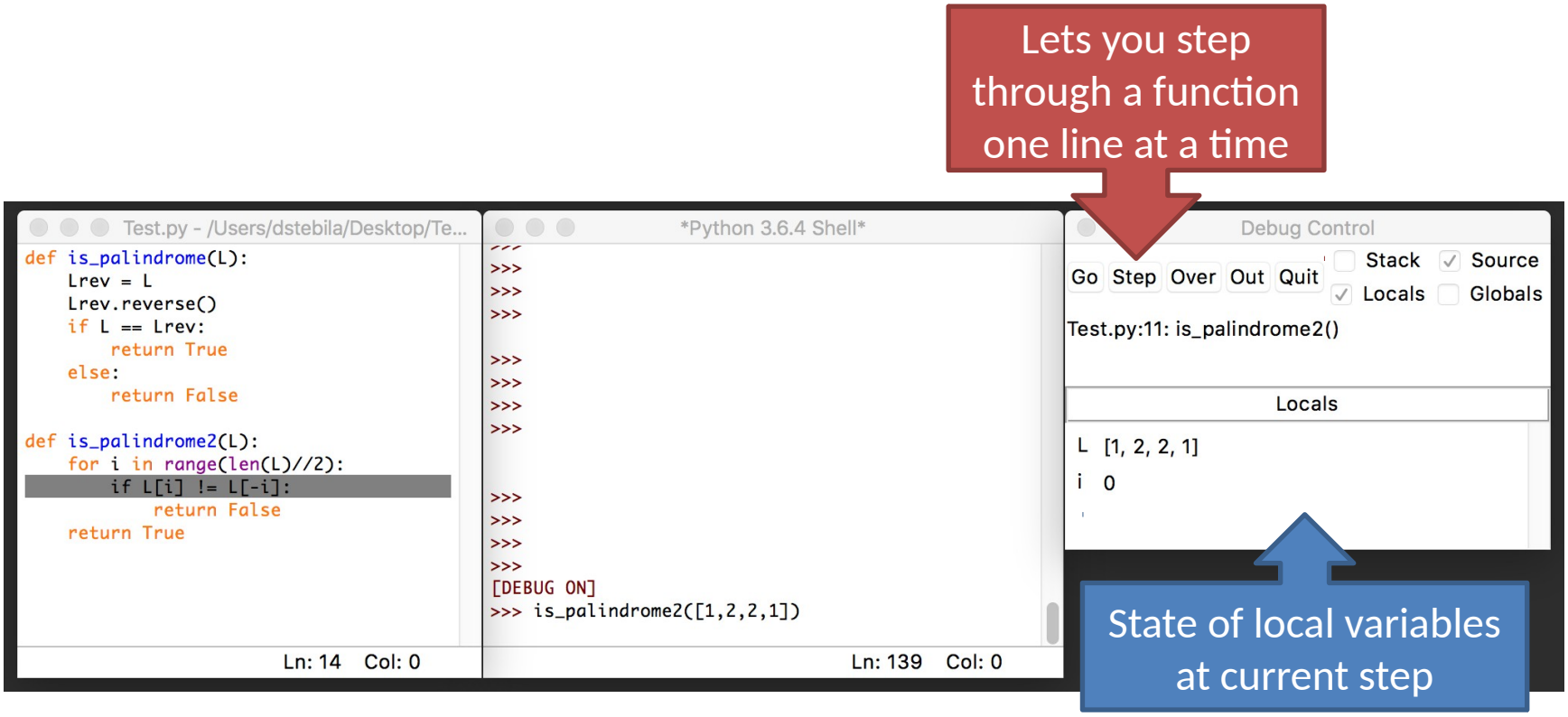
Debugger

- Integrates with an editor to visual inspect source code and internal state as a program is running
- Can step through each line of source code one at a time
- Can set **breakpoints**: run the source code until it reaches a certain line of source code, then pause to let the developer inspect it

IDLE

- Integrated development environment (IDE) for Python
- Built-in to all Python installations
- Run by typing
  - idle3

Debugging with IDLE



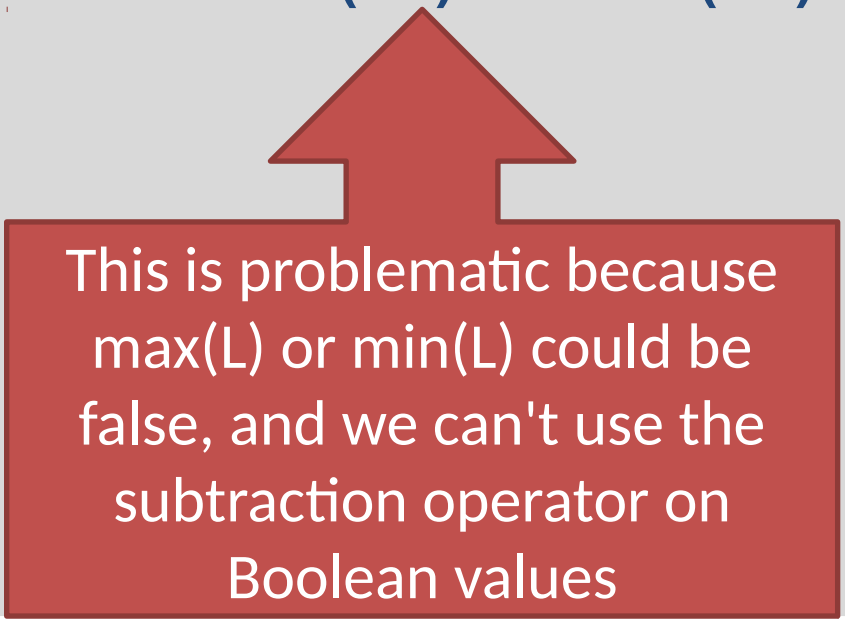
EXCEPTIONS

Something to think about

- Suppose you want to create a function that returns the maximum number in a list
- What should the function return if you input an empty list?
  - 0?
    - Not a good idea… 0 wasn't actually in the list
  - -infinity?
  - nan? ("Not a number")
  - False?
    - Would need to check for these results every time

If max and min returned False

```
def bound(L):
    """Find difference between
       max and min in list L"""
    return max(L)-min(L)
```



If max and min returned False

```
def bound(L):
    """Find difference between
       max and min in list L"""
    a = max(L)
    if (a == False): return
False
    b = min(L)
    if (b == False): return
False
    return a-b
```

Similarly...

- What should happen when you pass a string to an integer conversion?
- Should it give 0? That's not what was actually entered?
- False? That's not an

```
>>> a = int(input("Enter a number: "))
Enter a number: potato
```

Exceptions

- Exceptions allow Python to deal with situations where it tries to execute a statement that isn't well defined or violates some condition
- Python **raises** an exception
  - This interrupts the normal flow of execution
- There can be an exception **handler** which tries to recover
- Or the exception can be **unhandled** in which case the program crashes

Unhandled exception

```
def doIt():
    a = int(input("Enter a number:
"))

>>> doIt()
Enter a number: potato
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in doIt
ValueError: invalid literal for int
```

Handled exception

```
def doIt2():
    try:
        a = int(input("Enter a number: "))
    except ValueError:
        print("You did not enter a valid number")

>>> doIt2()
Enter a number: potato
```

Handling exceptions

- Put the code that might cause an exception inside a "try" block
- Use one or more "except ErrorName" blocks to specify what to do if the code causes one of those exceptions

- Can also use a "finally" block runs afterward no matter what (whether an exception was raised or not)

Raising exceptions

- You can **raise** exceptions in your code if it enters a situation your code isn't intended to handle

```
def findFirstEven(L):
  """Returns first even number in list L.
     Raises ValueError if no even number in
L."""
  for x in L:
    if x % 2 == 0: return x
  raise ValueError
```

Common exceptions

- IndexError: when the index to a list/string/etc. is out of bounds
  - Example: L = [0,1,2] ; L[27]
- TypeError: when a function/operator is applied to an object of the inappropriate type
  - Example: L = [0,1,2] ; L + 3
- ValueError: when a function/operator is applied to an object of the right type but otherwise inappropriate
  - Example: L = [] ; max(L)

Common exceptions

- NotImplementedError: when a function has not been implemented
- RecursionError: when the maximum recursion depth has been exceeded
- RuntimeError: some error occurred that isn't classified elsewhere
- ZeroDivisionError
  - Example: 5/0

Assertions

```
assert a == b
assert c > 7
assert f(4) == False
```

- Raises an AssertionError if the expression evaluates to False
- Useful for unit testing and defensive programming