Data Science

Testing and Debugging

Risa Myers

Rice University

Testing and debugging

- Defensive programming – make your program robust
- Testing – make sure your program functions as expected
- Debugging – fixing your program when it doesn’t work as expected

Code writing strategies

- Make it work
- Make it work right
- Make it work fast

✓

⏱

Defensive programming I

- Write clear SPECIFICATIONS (docstring)
  - Describe expected inputs
  - Describe outputs
  - Explains transformations (and any side effects) of inputs to outputs

  A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.[1]

  ...

  The one-line docstring should NOT be a "signature" reiterating the function/method parameters (which can be obtained by introspection).

  ...

  consist of a summary line ... a more elaborate description.

[1] https://www.python.org/dev/peps/pep-0257/

Defensive programming II

- Write modular programs
  - Break code down into small, meaningful pieces
  - Reuse code

```
def convertToKelvin(degreesF):
    return (degreesF - 32) * 5/9 + 273.15

def convertToCelsius(degreesF):
    return (degreesF - 32) * 5/9

def convert(degreesF, toCelcius):
    if toCelcius:
        returnVal = \
         convertToCelsius(degreesF)
    else:
        returnVal = \
         convertToKelvin(degreesF)

    return returnVal
```

Defensive programming III

- Check input conditions
  - Ensure your inputs are valid
  - Saves you debugging time later

Defensive programming IV

- Comment your code
  - Helps future readers (including you & the graders) understand logic
  - Saves you debugging time later
  - If we remove all of your code, the reader should be able to read your comments and understand your logic
  - You should comment blocks (not necessarily lines) of code

Once the code is written

- How do you know it’s right?

Once the code is written

- How do you know it’s right?
- You TEST

Test classes

- Unit tests
  - Check each individual function
- Regression tests
  - Ensure no new bugs have been introduced by changes
  - Rerun all your existing test cases
- Integration tests
  - Test the overall program
  - Test all the component functions

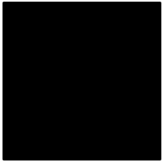
```
def convertToKelvin(degreesF):
    return (degreesF - 32) * 5/9 + 273.15

def convertToCelsius(degreesF):
    return (degreesF - 32) * 5/9

def convert(degreesF, toCelcius):
    if toCelcius:
        returnVal =  \
         convertToCelsius(degreesF)
    else:
        returnVal = \
         convertToKelvin(degreesF)
    return returnVal
```

Testing

1. “Black box” testing
2. “White box” testing

“Black box” testing

- Treat the code as an opaque machine
- Test paths through the SPECIFICATION
- DON’T look at the code
- Best written by someone other than the programmer
- Can be reused even if the implementation changes
- Check inputs against specifications
- Check output against specifications
- Try different types of inputs
    - Use your intuition
    - Use random inputs
- Consider boundary conditions for the inputs



Poll

- What are some test cases for this function?

```
def myFunction(a, b):
    '''
       this function checks to see if the value
       of a is strictly greater than the value of b
       If so, True is returned
       Otherwise, False is returned
    '''
    if a > b:
        return True
    else:
        return False
```

Boundary conditions

- Empty list
- 1 item list
- Negative numbers
- Zero
- Large numbers
- Items in (reverse) sorted order
- ...

“White box” testing

- Treat the code as an transparent machine
- Test paths through the CODE
- Run thru each loop 0, 1, and 2 times
- Test each path through each `IF` statement

```
def countDown(n):
    while n > 0:
        print(n)
        n = n - 1
```

Poll

- What are some test cases for this function?

```
def countDown(n):
    '''
       this function prints out the numbers from
       n to 0
    '''
    while n > 0:
        print(n)
        n = n - 1
```

Stubs

- A function substitute
- Can simulate the correct function behavior
- Used to test other functions

```
def containsFactor(n, f):
    return True
```

Types of errors I

- Syntax errors
  - Missing `:`, `(`, etc.
  - Interpreter / compiler informs you of these

```
3 +=  5
a[0)
```

Types of errors II

- Runtime errors
  - Trying to access past the end of a list
  - Accessing an invalid dictionary key
  - Calling a method not defined for an object
  - Referencing a non-existant object
  - Converting an object to an invalid type
  - Using different data types together
  - Infinite loops!

```
a = [ 4, 5, 6] a[5]

d = {1: ’apple’, 2:’banana’} d[3]
3.lower()


int(a)

’foo’ + 3
```

Types of errors III

- Specification errors
  - Your program does what it says it does, but it’s not what you want
  - Returns 0 instead of -1
  - Returns a dictionary instead of a list
- Logic errors
  - Your program doesn’t work as expected
  - Divide instead of multiply
  - Start with 1 instead of 0
- Arithmetic errors
  - Your program doesn’t work as expected wrt numbers
  - Divide by 0
  - Integer division

Types of errors IV

- Intermittent
  - Only occurs sometimes, even when the same test case is called
  - Often due to
    - Initialization errors
    - Side effects (e.g. changing a function input parameter)
- Persistent
  - Error happens every time a test case is called
  - Easier to debug

Dealing with errors

- You found an error
- How do you fix it?

Dealing with errors

- You found an error
- How do you fix it?
- You DEBUG your code

Debugging

- Can be time consuming
- Has a steep learning curve
- Can be frustrating

Debugging tools

- IDEs (pycharm, Spyder)
- Print statements
- ipython
- Think about what might be going on

  ... and test that theory

Print statements

- When you enter a function
- When you complete an action
- The input parameters
- The variables
- The outputs

```
def doesStuff(x):
    print(’x: ’, x)
    while x > 0:
        print(’while x: ’, x)
        x = x + 1
        print(’new x: ’, x)
    print(’return x: ’, x)
    return x
```

Code debugging strategies

- Take a break
- Explain the code to someone else (not in the course)

Try It! Debugging

```
# version 1
def Factorial (n):
    if n == 1 or n == 0:
        return 1
    else:
        return n * Factorial (n + 1)
```

```
# version 2
def Factorial (n):
    if n === 1:
        return 1
    else:
        return n * Factorial (n - 1)
```

```
# version 3
def Factorial (n):
    if n == 1 or n == 0:
        return 1
    else:
        return n * Factoral (n - 1)
```

Questions?

? How can we use what we learned today?

? What do we know now that we didn’t know before?