Module 8

Testing

Basic Concepts

- Some important terms
  - **Bug**: Error in a program. (Always expect them!)
  - **Debugging**: Process of finding & removing bugs
  - **Testing**: Process of analyzing & running a program
- Testing is a common way to search for bugs
  - However, it is not the only way
  - And it does not address how to remove them
- Good debugging starts with testing

Test Cases: Searching for Errors

- Testing is done with **test cases**
  - An input, together with an expected output
  - Input is the one (or more) argument(s)
  - Output is what is returned
  - Or what side-effect the procedure causes
- A list of test cases is **testing plan**
  - Similar to what we did when reading specs

Testing Plan: A Case Study

```
def number_vowels(w):
    """
    Returns: number of vowels in string w.

    Vowels are defined to be 'a','e','i','o', and 'u'. 'y' is a vowel if it is
    not at the start of the word.

    Repeated vowels are counted separately.  Both upper case and
    lower case vowels are counted.

    Examples: ....

    Parameter w: The text to check for vowels
    Precondition: w string w/ at least one letter and only letters
    """
```

Testing Plan: A Case Study

```
def number_vowels(w):
    """
    Returns: number of vowels

    Vowels are defined to be 'a
    not at the start of the wor

    Repeated vowels are counted separately.  Both upper case and
    lower case vowels are counted.

    Examples: ....

    Parameter w: The text to check for vowels
    Precondition: w string w/ at least one letter and only letters
    """
```

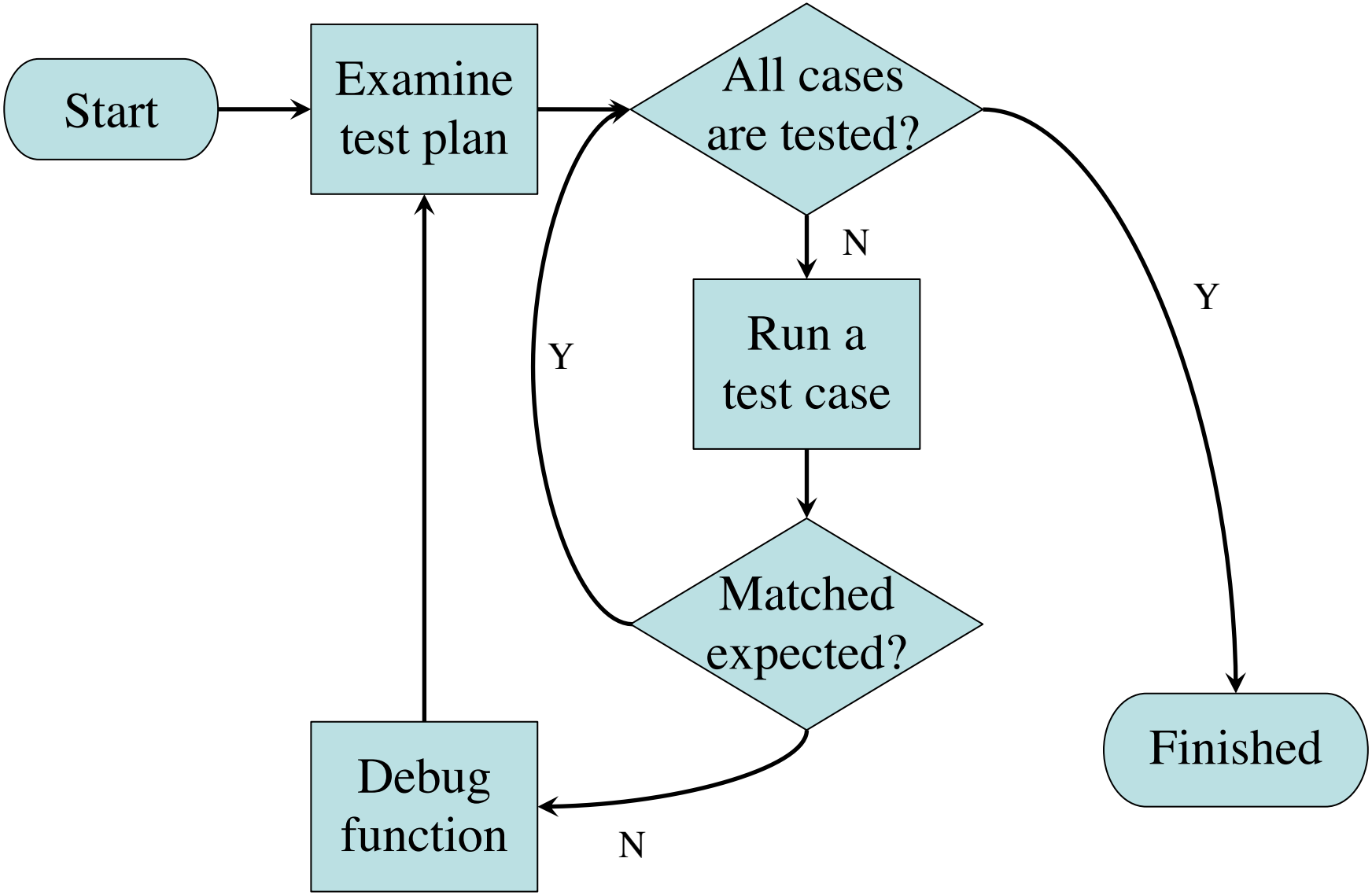
INPUT	OUTPUT
'hat'	1
'heat'	2
sky'	1
'year'	2
'xxx'	0

Recall: Workflow for this Course

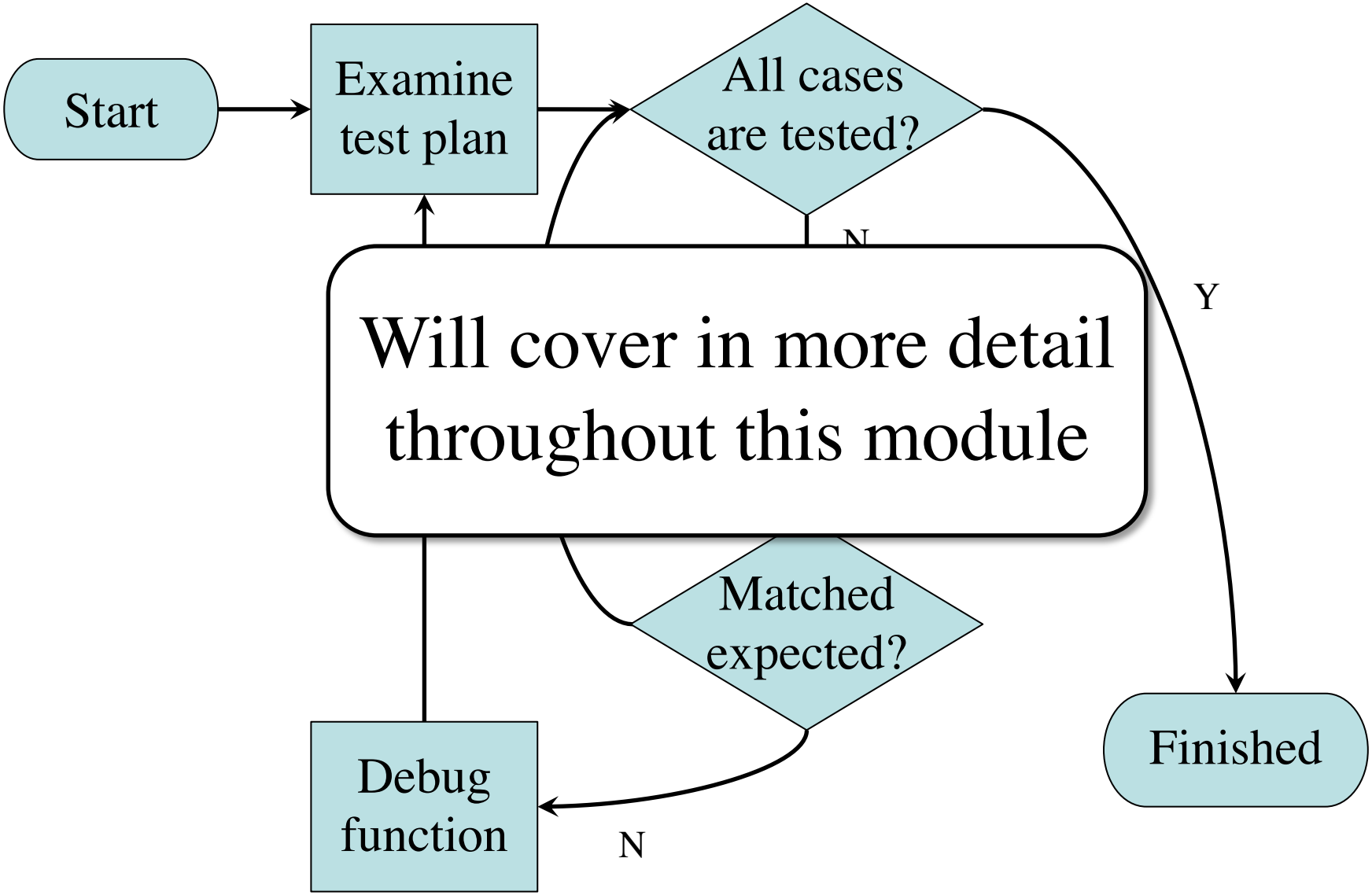
1. Write a procedure (function) in a module
2. Open up the Terminal
3. Move to the directory with this file
4. Start Python (type `python`)
5. Import the module
6. **Call the procedure (function)**

Testing!

How to Test a Function



How to Test a Function



Testing Plan: A Case Study

```
def number_vowels(w):
    """
    Returns: number of vowels in string w.

    Vowels are defined to be 'a','e','i','o', and 'u'. 'y' is a vowel if it is
    not at the start of the word.

    Repeated vowels are counted sepa
    lower case vowels are counted.

    Examples: ....

    Parameter w: The text to check for vowels
    Precondition: w string w/ at least one letter and only letters
    """
```

How many tests is enough?

Representative Tests

- We cannot test all possible inputs
  - "Infinite" possibilities (strings arbitrary length)
  - Even if finite, way too many to test
- Limit to tests that are **representative**
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- This is an art, not a science
  - If easy, no one would ever have bugs
  - Learn with much practice (and why teach early)

Representative Tests

Simplest case first!

A little complex

"Weird" cases

Representative Tests for `number_vowels(w)`

- Word with just one vowel
  - For each possible vowel!
- Word with multiple vowels
  - Of the same vowel
  - Of different vowels
- Word with only vowels
- Word with no vowels

How Many "Different" Tests Are Here?

number_vowels(w)

INPUT	OUTPUT
'hat'	1
'charm'	1
'bet'	1
'beet'	2
'beetle'	3

A: 2
B: 3 **CORRECT(ISH)**
C: 4
D: 5
E: I do not know

- If in doubt, just add more tests
- You are (rarely) penalized for too many tests

The Rule of Numbers

- When testing the numbers are 1, 2, and 0
- **Number 1**: The simplest test possible
  - If a complex test fails, what was the problem?
  - **Example**: Word with just one vowels
- **Number 2**: Add more than was expected
  - **Example**: Multiple vowels (all ways)
- **Number 0**: Make something missing
  - **Example**: Words with no vowels

HOWEVER

- **NEVER** test a violation of precondition
  - Why? You have no idea what happens
  - Unspecified means no guarantees at all
  - So you have no correct answer to compare
- **Example**: `'bcd'` okay, but `'12a'` is bad.
- This can effect the rule of 1, 2, and 0
  - Precondition may disallow the rule
  - **Example**: a string with at least one value

Test Script: A Special Kind of Script

- Right now to test a function we do the following
  - Start the Python interactive shell
  - Import the module with the function
  - Call the function several times to see if it is okay
- But this is incredibly time consuming!
  - Have to quit Python if we change module
  - Have to retype everything each time
- What if we made a **second** Python module/script?
  - This module/script tests the first one

Test Script: A Special Kind of Script

- A test script is designed to test another module
  - It imports the other module (so it can access it)
  - It defines one or more test cases
  - It calls the function on each input
  - It compares the result to an expected output
- Doesn't do much if everything is fine
- If wrong, it prints out helpful information
  - What was the case that failed?
  - What was the wrong answer given?

Testing with assert_equals

- Testing uses a special function:

```
def assert_equals(expected,received):
    """Quit program if expected, received differ"""
```

- Provided by the introcs module
  - Special module used for this course
  - Documentation is on course web page
  - Also contains useful string functions
  - And other functions beyond course scope

Running Example

- The following function has a bug:

```
def last_name_first(n):
    """Returns: copy of <n> but in the form <last-name>, <first-name>

    Precondition: <n> is in the form <first-name> <last-name>
    with one or more blanks between the two names"""
    end_first = n.find(' ')
    first = n[:end_first]
    last  = n[end_first+1:]
    return last+', '+first
```

Look at precondition when choosing tests

- Representative Tests:
  - last_name_first('Walker White') give 'White, Walker'
  - last_name_first('Walker White') gives 'White, Walker'

Testing last_name_first(n)

```
import name                 # The module we want to test
import introcs              # Includes the test procedures

# First test case
result = name.last_name_first('Walker White')
introcs.assert_equals('White, Walker', result)

# Second test case
result = name.last_name_first('Walker          White')
introcs.assert_equals('White, Walker', result)

print('Module name is working correctly')
```

Testing last_name_first(n)

```
import name                    # The module we want to test
import introcs                 # Includes the test procedures

# First test case
result = name.last_name_first('Walker White')
introcs.assert_equals('White, Walker', result)

# Second test case
result = name.last_name_first('Walker          White')
introcs.assert_equals('White, Walker', result)

print('Module name is working correctly')
```

Actual Output

Input

Expected Output

Testing last_name_first(n)

```
import name                 # The module we want to test
import introcs              # Includes the test procedures

# First test case
result = name.last_name_first('Walker White')
introcs.assert_equals('White, Walker', result)

# Second test case
result = name.last_name_first('Walker          White')
introcs.assert_equals('White, Walker', result)

print('Module name is working correctly')
```

Quits Python if not equal

Message will print out only if no errors.

Testing last_name_first(n)

```
import name                 # The module we want to test
import introcs              # Includes the test procedures

# First test case
result = name
introcs.asser

# Second test
result = name.
introcs.assert_equals( White, Walker', result)

print('Module name is working correctly')
```

Finish example with number_of_vowels

Using Test Procedures

- In the real world, we have a lot of test cases
  - I wrote 20000+ test cases for a C++ game library
  - This is not all one function!
  - You need a way to cleanly organize them
- **Idea**: Put test cases inside another procedure
  - Each function tested gets its own procedure
  - Procedure has test cases for that function
  - Also some print statements (to verify tests work)

Running Example

- The following function has a bug:

```
def last_name_first(n):
    """Returns: copy of <n> but in the form <last-name>, <first-name>

    Precondition: <n> is in the form <first-name> <last-name>
    with one or more blanks between the two names"""
    end_first = n.find(' ')
    first = n[:end_first]
    last  = n[end_first+1:]
    return last+', '+first
```

Look at precondition when choosing tests

- Representative Tests:
  - last_name_first('Walker White') give 'White, Walker'
  - last_name_first('Walker White') gives 'White, Walker'

Test Procedure

```
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    print('Testing function last_name_first')
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    introcs.assert_equals('White, Walker', result)

# Execution of the testing code
test_last_name_first()
print('Module name is working correctly')
```

Actual file has 2 funcs

Test Procedure

```
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    print('Testing function last_name_first')
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    introcs.assert_equals('White, Walker', result)

# Execution of the testing code
test_last_name_first()
print('Module name is working correctly')
```

Actual file has 2 funcs

No tests happen if you forget this

Test Procedure

```
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    print('Testing function last_name_first')
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    introcs.assert_equals('White, Walker', result)

# Execution of the testing code
# test_last_name_first()
print('Module name is working correctly')
```

Actual file has 2 funcs

Can remove to disable test

Testing last_name_first(n)

```
# test procedure
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    introcs.assert_equals('White, Walker', result)
```

Call function on test input

Compare to expected output

```
# Script code
test_last_name_first()
print('Module name is working correctly')
```

Call test procedure to activate the test

Types of Testing

Black Box Testing

- Function is "opaque"
  - Test looks at what it does
  - **Fruitful**: what it returns
  - **Procedure**: what changes
- **Example**: Unit tests
- **Problems**:
  - Are the tests everything?
  - What caused the error?

White Box Testing

- Function is "transparent"
  - Tests/debugging takes place inside of function
  - Focuses on where error is
- **Example**: Use of `print`
- **Problems**:
  - Much harder to do
  - Must remove when done

Types of Testing

Black Box Testing

- Function is "opaque"
  - Test looks at what it does
  - rns
  - nges
- **Ex**
- **Problems**:
  - Are the tests everything?
  - What caused the error?

Works on functions you did not define

White Box Testing

- Function is "transparent"
  - es
  - on
  - or is
- **Ex** nt
- **Problems**:
  - Much harder to do
  - Must remove when done

Can actually find the bug in function

Finding the Error

- Unit tests cannot find the source of an error
- Idea: "Visualize" the program with print statements

```
def last_name_first(n):
    """Returns: copy of <n> in form <last>, <first>"""
    end_first = n.find(' ')
    print(end_first)
    first = n[:end_first]
    print(str(first))
    last  = n[end_first+1:]
    print(str(last))
    return last+', '+first
```

Print variable after each assignment

Run Demo

How to Use the Results

- Goal of **white box testing** is error location
  - Want to identify the **exact line** with the error
  - Then you look real hard at line to find error
  - What you did in earlier assessment
- But similar approach to **black box testing**
  - At each line you have **expected** print result
  - Compare it to the **received** print result
  - Line before first mistake is *likely* the error

Finding the Error

- Unit tests cannot find the source of an error
- Idea: "Visualize" the program with print statements

```
def last_name_first(n):
    """Returns: copy of <n> in form <last>, <first>"""
    end_first = n.find(' ')
    print(end_first)
    first = n[:end_first]
    print(str(first))
    last  = n[end_first+1:]
    print(str(last))
    return last+', '+first
```

Do as little as possible on each line of code

Finding the Error

- Unit tests cannot find the source of an error
- Idea: "Visualize" the program with print statements

```
def last_name_first(n):
    """Returns: copy of <n> in form <last>, <first>"""
    end_first = n.find(' ')
    print('space at '+end_first)
    first = n[:end_first]
    print('first is '+str(first))
    last  = n[end_first+1:]
    print('last is '+str(last))
    return last+', '+first
```

Print variable after each assignment

Optional: Annotate value to make it easier to identify

Warning About Print Statements

- Must remove them when you are done
  - Not part of the specification (violation)
  - Slow everything down unnecessarily
  - App Store will reject an app with prints
- But you might want them again later
  - **Solution**: “comment them out”
  - Can uncomment later if need them