Lecture 5

Specifications & Testing

Announcements For This Lecture

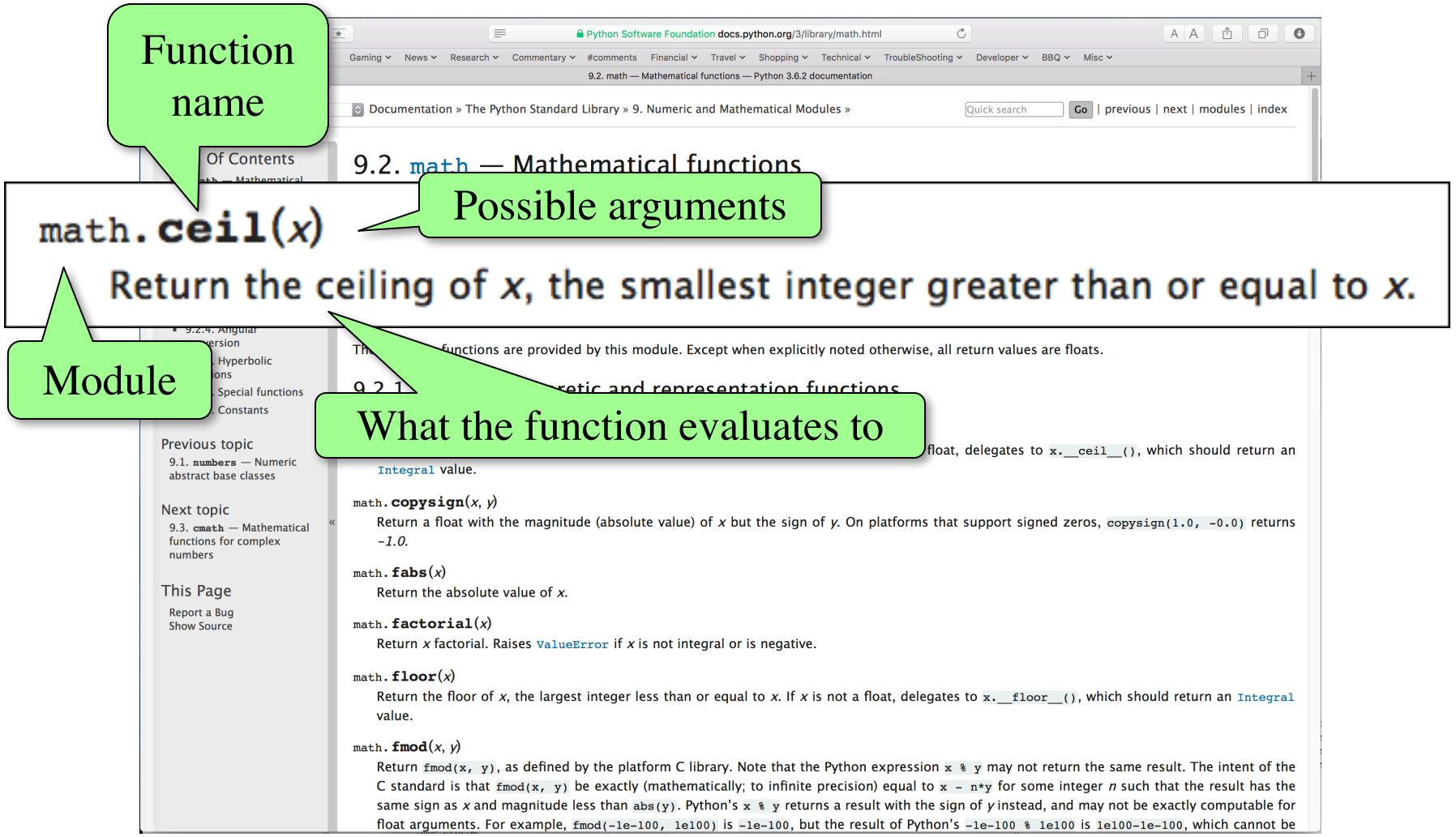
Last Call

- Acad. Integrity Quiz
- E-mails sent out Sunday
- Will drop tomorrow

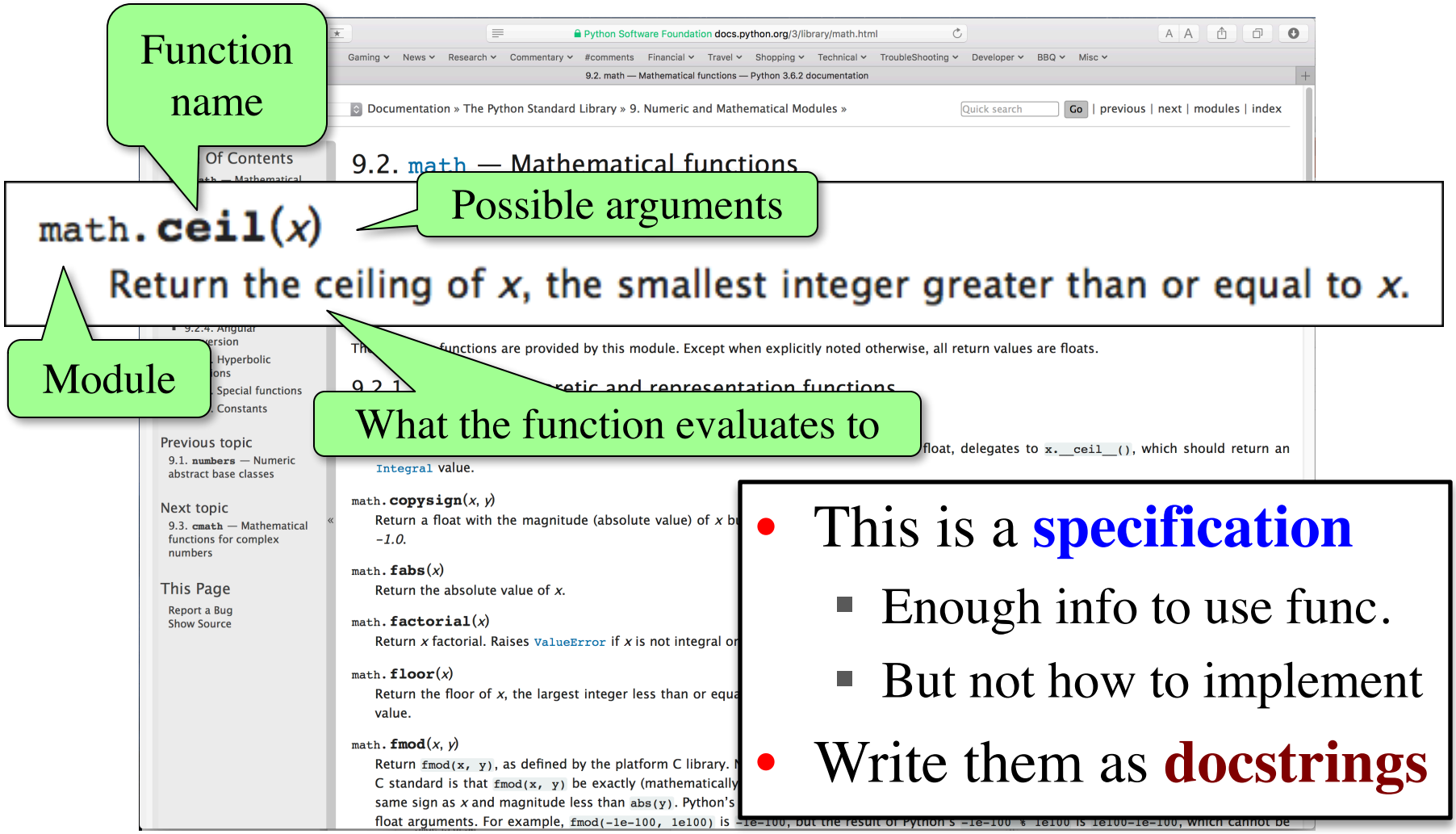
Assignment 1

- Posted on web page
  - Due Thu, Sep. 21nd
  - Due in place of Lab 4
  - Revise until correct
- Can work in pairs
  - One submission for pair
  - Must work **together**

Recall: The Python API



Recall: The Python API



- This is a **specification**
  - Enough info to use func.
  - But not how to implement
- Write them as **docstrings**

Anatomy of a Specification

One line description, followed by blank line

```
def greet(n):
    """Prints a greeting to the name n

    Greeting has format 'Hello <n>!'
    Followed by conversation starter.

    Parameter n: person to greet
    Precondition: n is a string"""
    print('Hello '+n+'!')
    print('How are you?')
```

Anatomy of a Specification

```
def greet(n):
    """Prints a greeting to the name n

    Greeting has format 'Hello <n>!'
    Followed by conversation starter.

    Parameter n: person to greet
    Precondition: n is a string"""
    print('Hello '+n+'!')
    print('How are you?')
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Anatomy of a Specification

```
def greet(n):
    """Prints a greeting to the name n

    Greeting has format 'Hello <n>!'
    Followed by conversation starter.

    Parameter n: person to greet
    Precondition: n is a string"""
    print('Hello '+n+'!')
    print('How are you?')
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Parameter description

Anatomy of a Specification

```
def greet(n):
    """Prints a greeting to the name n

    Greeting has format 'Hello <n>!'
    Followed by conversation starter.

    Parameter n: person to greet
    Precondition: n is a string"""
    print('Hello '+n+'!')
    print('How are you?')
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Parameter description

Precondition specifies assumptions we make about the arguments

Anatomy of a Specification

```
def to_centigrade(x):
    """Returns: x converted to centigrade

    Value returned has type float.

    Parameter x: temp in fahrenheit
    Precondition: x is a float"""
    return 5*(x-32)/9.0
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Parameter description

Precondition specifies assumptions we make about the arguments

Anatomy of a Specification

```
def to_centigrade(x):
    """Returns: x converted to centigrade

    Value returned has type float.

    Parameter x: temp in fahrenheit
    Precondition: x is a float"""
    return 5*(x-32)/9.0
```

"Returns" indicates a fruitful function

More detail about the function. It may be many paragraphs.

Parameter description

Precondition specifies assumptions we make about the arguments

Preconditions

- Precondition is a promise
  - If precondition is true, the function works
  - If precondition is false, no guarantees at all
- Get **software bugs** when
  - Function precondition is not documented properly
  - Function is used in ways that violates precondition

```
>>> to_centigrade(32.0)
0.0
>>> to_centigrade(212)
100.0
```

Preconditions

- Precondition is a promise
  - If precondition is true, the function works
  - If precondition is false, no guarantees at all
- Get **software bugs** when
  - Function precondition is not documented properly
  - Function is used in ways that violates precondition

```
>>> to_centigrade(32.0)
0.0
>>> to_centigrade(212)
100.0
>>> to_centigrade('32')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "temperature.py", line 19 ...
TypeError: unsupported operand type(s)
for -: 'str' and 'int'
```

Precondition violated

Test Cases: Finding Errors

- **Bug**: Error in a program. (Always expect them!)
- **Debugging**: Process of finding bugs and removing them.
- **Testing**: Process of analyzing, running program, looking for bugs.
- **Test case**: A set of input values, together with the expected output.

Get in the habit of writing test cases for a function from the function's specification —even *before* writing the function's body.

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

    Precondition: w string w/ at least one letter and only letters"""
    pass  # nothing here yet!
```

Test Cases: Finding Errors

- **Bug**: Error in a program. (Always
- **Debugging**: Process of finding bug
- **Testing**: Process of analyzing, runn
- **Test case**: A set of input values, to

Get in the habit of writing test case
function's specification —even *be*

Some Test Cases

- number_vowels('Bob')
  Answer should be 1
- number_vowels('Aeiuo')
  Answer should be 5
- number_vowels('Grrr')
  Answer should be 0

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

    Precondition: w string w/ at least one letter and only letters"""
    pass  # nothing here yet!
```

Representative Tests

- Cannot test all inputs
  - "Infinite" possibilities
- Limit ourselves to tests that are **representative**
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- An art, not a science
  - If easy, never have bugs
  - Learn with much practice

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
C: 4
D: 5
E: I do not know

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 never penalized for too many tests

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
```

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

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'

Unit Test: 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

Unit Test: A Special Kind of Script

- A unit test is a script that tests another module
  - It **imports the other module** (so it can access it)
  - It **imports the `cornell` module** (for testing)
  - It **defines one or more test cases**
    - A representative input
    - The expected output
- The test cases use the `cornell` function

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

Testing last_name_first(n)

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

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

# Second test case
result = name.last_name_first('Walker          White')
cornell.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 cornell              # Includes the test procedures

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

# Second test case
result = name.last_name_first('Walker          White')
cornell.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 cornell              # Includes the test procedures


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


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


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

Quits Python if not equal

Message will print out only if no errors.

Using Test Procedures

- In the real world, we have a lot of test cases
  - I wrote 20000+ test cases for a C++ game library
  - 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)
- Turn tests on/off by calling the test procedure

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')
    cornell.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    cornell.assert_equals('White, Walker', result)

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

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')
    cornell.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker          White')
    cornell.assert_equals('White, Walker', result)

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

No tests happen if you forget this