http://www.cs.cornell.edu/courses/cs1110/2019sp

Lecture 6: Specifications & Testing

(Sections 4.9, 9.5)

CS 1110

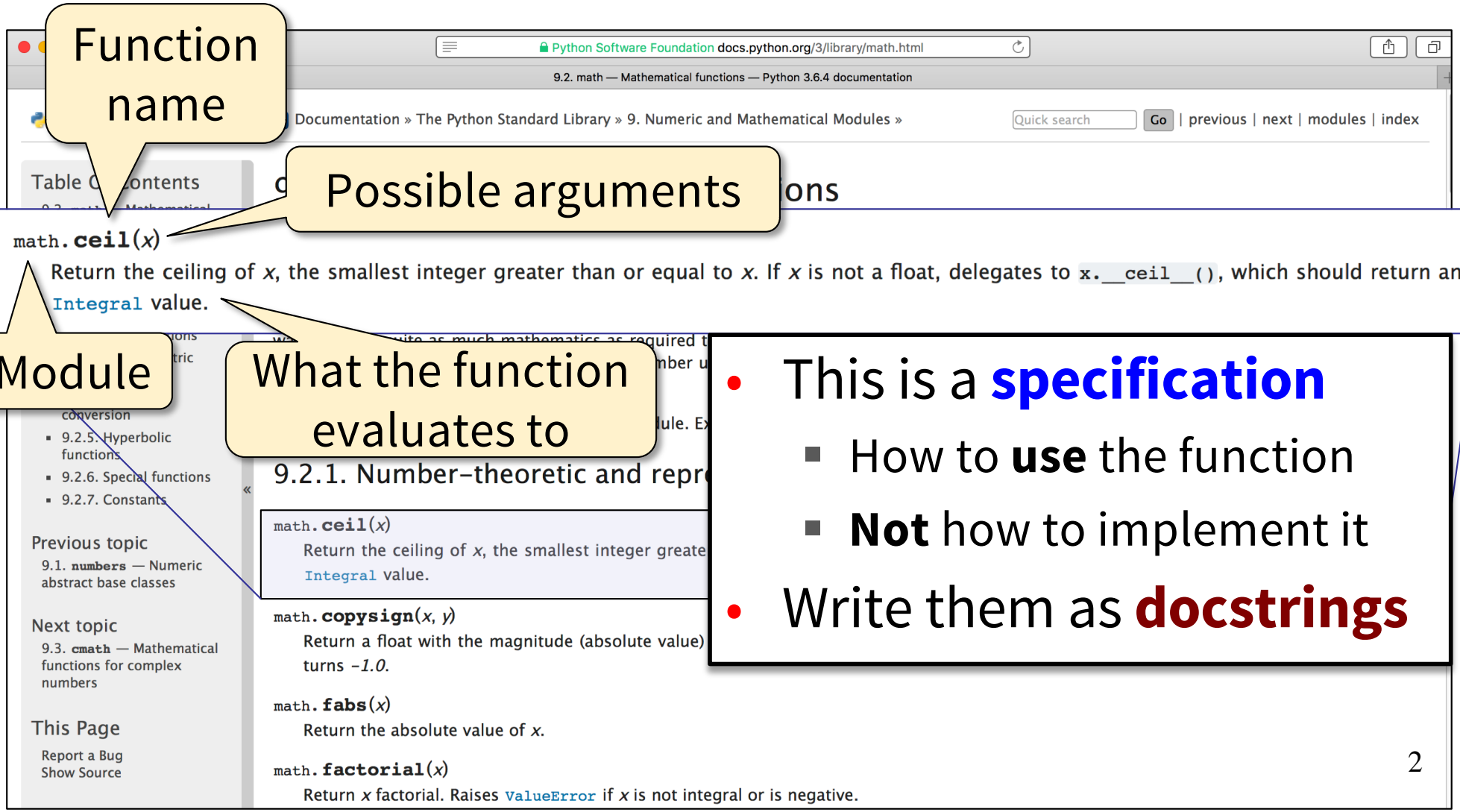
Introduction to Computing Using Python

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Recall the Python API

https://docs.python.org/3/library/math.html



Anatomy of a Specification

```
def greet(name):
    """Prints a greeting to person name
    followed by conversation starter.

    <more details could go here>

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

Short description, followed by blank line

As needed, more detail in 1 (or more) paragraphs

Parameter description

Precondition specifies assumptions we make about the arguments

Anatomy of a Specification

```
def get_campus_num(phone_num):
    """Returns the on-campus version
    of a 10-digit phone number.

    Returns:  str of form "X-XXXX"

    phone_num: number w/area code
    Precondition: phone_num is a 10
    digit string of only numbers"""
    return phone_num[5]+"-"+phone_num[6:10]
```

Short description, followed by blank line

Information about the return value

Parameter description

Precondition specifies assumptions we make about the arguments

A Precondition Is a Contract

- Precondition is met:
  The function will work!
- Precondition not met?
  Sorry, no guarantees…

Software bugs occur if:

- Precondition is not documented properly
- Function use violates the precondition

```
>>> get_campus_num("6072554444")
'5-4444'
>>> get_campus_num("6072531234")
'3-1234'
>>> get_campus_num(6072531234)
Traceback (most recent call last):
 File "<stdin>", line 1, in<module>
 File "/Users/bracy/cornell_phone.py", line 12, in get_campus_num
   return phone_num[5]+"-"+phone_num[6:10]
TypeError: 'int' object is not subscriptable
>>> get_campus_num("607-255-4444")
'5-5-44'
```

Precondition violated: **error!**

Precondition violated: **no error!**

Question: Which is worse?

Both **#1** and **#2** violate a precondition.

Which is worse?

A. #1

B. #2

C. They are equally bad.

D. Come on. Neither is *so* bad.

E. I don't know

#1

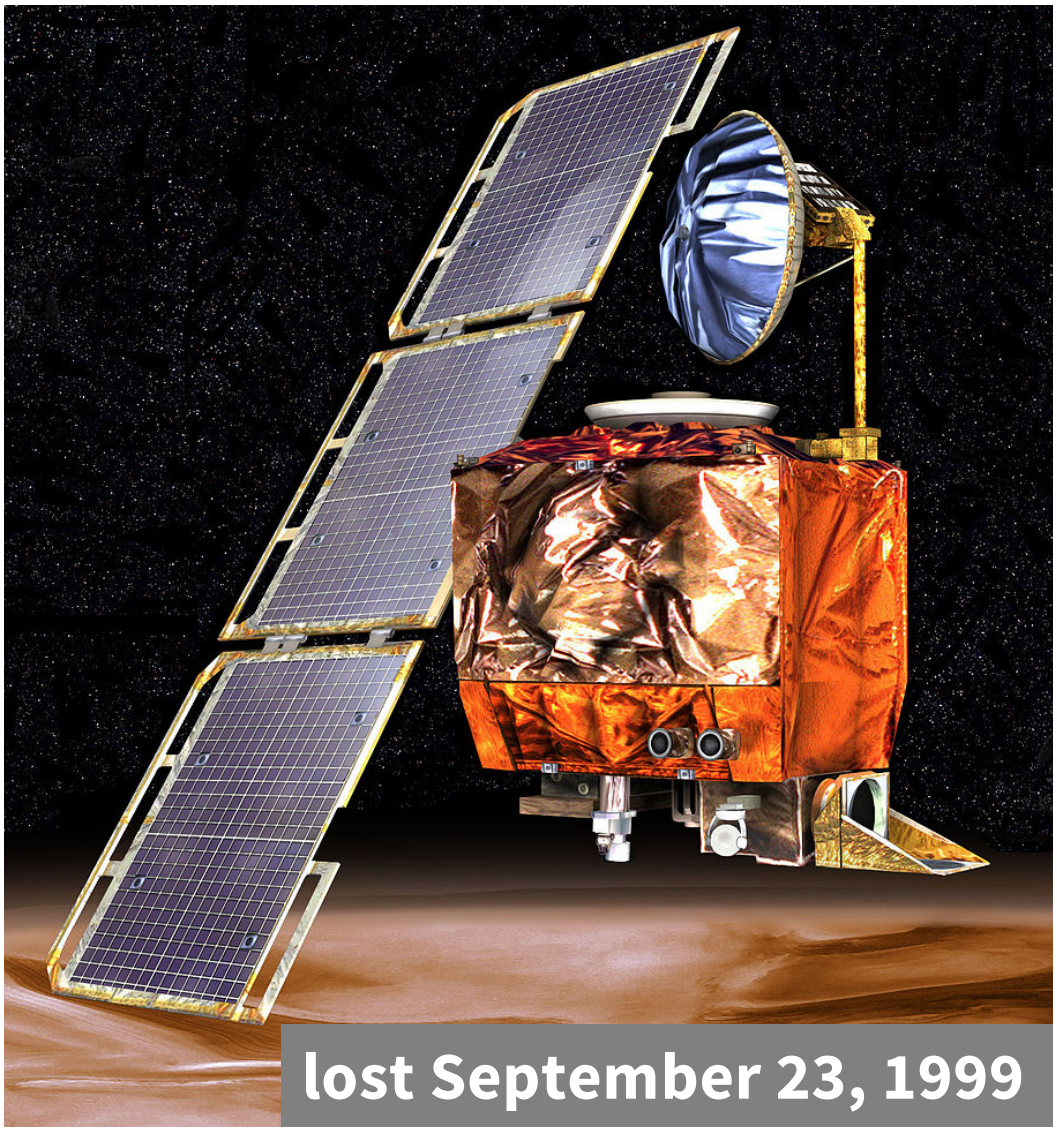
```
>>> get_campus_num(6072531234)
Traceback (most recent call last):
 File "<stdin>", line 1, in<module>
 File "/Users/bracy/cornell_phone.py", line 12, in get_campus_num
   return phone_num[5]+"-"+phone_num[6:10]
TypeError: 'int' object is not subscriptable
```

#2

```
>>> get_campus_num("607-255-4444")
'5-5-44'
```

NASA Mars Climate Orbiter

"NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation..."



Sources: Wikipedia & CNN

Preconditions Make Expectations Explicit

In American terms:

Preconditions help assign blame.

Something went wrong.

Did you use the function wrong?

OR

Was the function implemented/specified wrong?

Basic Terminology

- **Bug**: an error in a program. Expect them!
  - Conceptual & implementation
- **Debugging**: the process of finding bugs and removing them
- **Testing**: the process of analyzing and running a 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 its specification
– even *before* writing the function itself!

Test Cases help you find errors

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

    word: a string with at least one letter and only letters"""
    pass  # nothing here yet!
```

Some Test Cases

- vowel_count('Bob')
  Expect: 1
- vowel_count('Aeiuo')
  Expect: 5
- vowel_count('Grrr')
  Expect: 0

More Test Cases

- vowel_count('y')
  Expect: 0? 1?
- vowel_count('Bobo')
  Expect: 1? 2?

Test Cases can help you find errors in the **specification** as well as the implementation.

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 `vowel_count(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

What should I be testing?

Common Cases: typical usage (see previous slide)

Edge Cases: live at the boundaries

- Target location in list: first, middle, last elements
- Input size: 0,1,2, many (length of lists, strings, etc.)
- Input Orders: max(big, small), max(small, big)…
- Element values: negative/positive, zero, odd/even
- Element types: int, float, str, *etc*.
- Expected results: negative, 0, 1, 2, many

Not all categories/cases apply to all functions.

Use your judgement!

Representative Tests Example

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

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

Look at precondition when choosing tests

Representative Tests:

- last_name_first('Maya Angelou') Expects: 'Angelou, Maya'
- last_name_first('Maya Angelou') Expects: 'Angelou, Maya'

Debugging with Test Cases (Question)

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

       full_name: has the form <first-name> <last-name>
       with one or more blanks between the two names"""“
       #get index of space after first name
1      space_index = full_name.find(' ')
       #get first name
2      first = full_name[:space_index]
       #get last name
3      last  = full_name[space_index+1:]
       #return “<last-name>, <first-name>”
4      return last+', '+first
```

Which line is “wrong”?

A: Line 1

B: Line 2

C: Line 3

D: Line 4

E: I do not know

- last_name_first('Maya Angelou’) gives 'Angelou, Maya'
- last_name_first('Maya Angelou’) gives ' Angelou, Maya'

Debugging with Test Cases (Solution)

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

       full_name: has the form <first-name> <last-name>
       with one or more blanks between the two names"""“
       #get index of space after first name
1      space_index = full_name.find(' ')
       #get first name
2      first = full_name[:space_index]
       #get last name
3      last  = full_name[space_index+1:]
       #return “<last-name>, <first-name>”
4      return last+', '+first
```

Which line is “wrong”?

A: Line 1

B: Line 2

C: Line 3 **CORRECT**

D: Line 4

E: I do not know

- last_name_first('Maya Angelou’) gives 'Angelou, Maya'
- last_name_first('Maya Angelou’) gives ' Angelou, Maya'

Motivating a Unit Test

- Right now to test a function, we:
  - Start the Python interactive shell
  - Import the module with the function
  - Call the function several times to see if it works right
- Super time consuming! ☹
  - Quit and re-enter python every time we change module
  - Type and retype…
- What if we wrote a script to do this ?!

Unit Test: A Special Kind of Script

- A unit test is a script that tests another module. It:
  - **Imports the module to be tested** (so it can access it)
  - **Imports `introcs` module** (for testing)
  - **Defines one or more test cases** that each include:
    - A representative input
    - The expected output
  - Test cases use the `introcs` function:

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

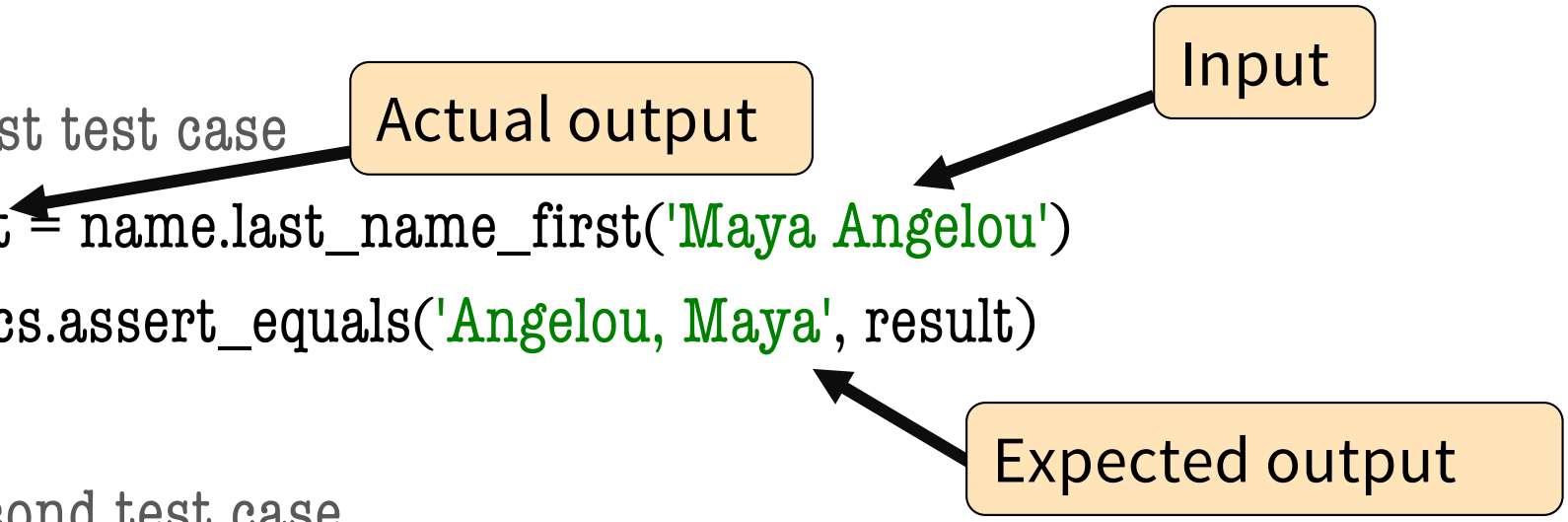
Testing last_name_first(full_name)

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

# First test case
result = name.last_name_first('Maya Angelou')
introcs.assert_equals('Angelou, Maya', result)

# Second test case
result = name.last_name_first('Maya              Angelou')
introcs.assert_equals('Angelou, Maya', result)

print('All tests of the function last_name_first passed')
```



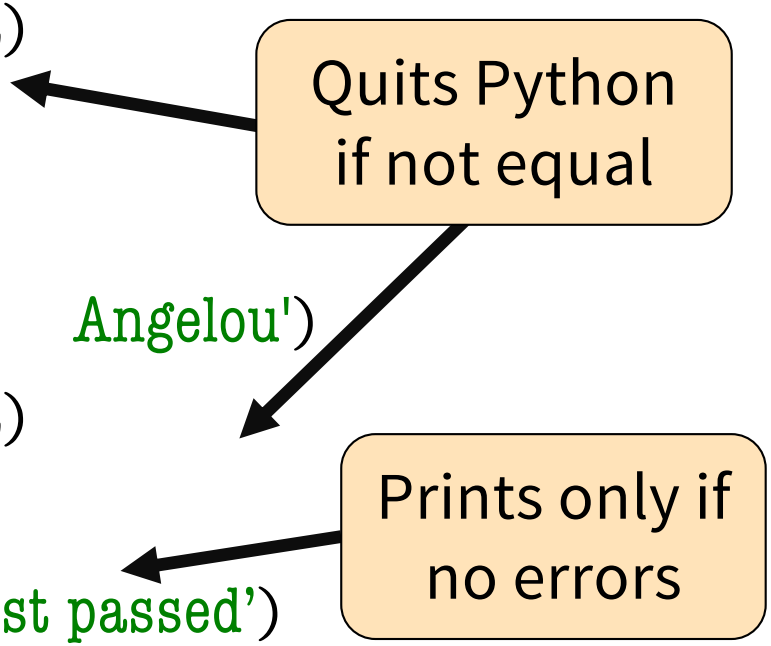
Testing last_name_first(full_name)

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


# First test case
result = name.last_name_first('Maya Angelou')
introcs.assert_equals('Angelou, Maya', result)


# Second test case
result = name.last_name_first('Maya                Angelou')
introcs.assert_equals('Angelou, Maya', result)


print('All tests of the function last_name_first passed')
```



Organizing your Test Cases

- We often have a lot of test cases
  - Common at (good) companies
  - Need a way to cleanly organize them

Idea: Bundle all test cases into a single test!

- One **high level test** for each function you test
- High level test performs **all** test cases for function
- Also uses some print statements (for feedback)

One Test to Rule them All

```
def test_last_name_first():
    """Calls all the tests for last_name_first"""
    print('Testing function last_name_first')
    # Test 1
    result = name.last_name_first('Maya Angelou')
    introcs.assert_equals('Angelou, Maya', result)
    # Test 2
    result = name.last_name_first('Maya              Angelou')
    introcs.assert_equals('Angelou, Maya', result)

# Execution of the testing code
test_last_name_first()
print('All tests of the function last_name_first passed')
```

No tests happen if you forget this

How to debug

Do **not** ask:

"Why doesn't my code do what I want it to do?"

Instead, ask:

"What is my code doing?"

Two ways to inspect your code:

1. Step through your code, drawing pictures (or *use python tutor!*)
2. Use print statements

Take a look in the python tutor!

```
def last_name_first(full_name):
    <snip out comments for ppt slide>
     #get index of space
    space_index = full_name.find(' ')
    #get first name
    first = full_name[:space_index]
    #get last name
    last  = full_name[space_index+1:]
    #return "<last-name>, <first-name>"
    return last+', '+first

last_name_first("Maya Angelou")
```

Pay attention to:

- Code you weren't 100% sure of as you wrote it
- Code relevant to the failed test case

Using print statement to debug

```
def last_name_first(full_name):
    print("full_name = "+full_name)
    #get index of space
    space_index = full_name.find(' ')
    print("space_index = "+ str(space_index))
    #get first name
    first = full_name[:space_index]
    print("first = "+ first)
    #get last name
    last  = full_name[space_index+1:]
    #return "<last-name>, <first-name>"
    print("last = "+ last)
    return last+', '+first
```

Sometimes this is your only option, but it does make a mess of your code, and introduces cut-n-paste errors.

How do I print this?