http://www.cs.cornell.edu/courses/cs1110/2021sp

Lecture 6: Specifications & Testing

(Sections 4.9, 9.5)

CS 1110

Introduction to Computing Using Python

Revisions made during/after lecture appear in **orange**

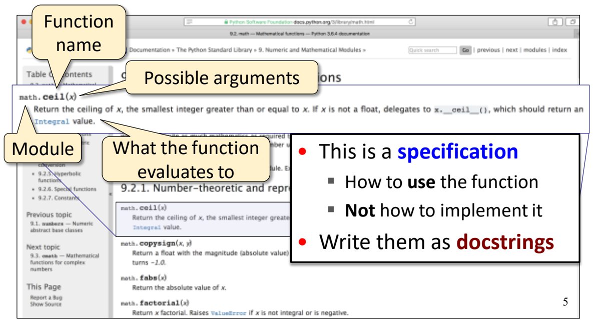
[E. Andersen, A. Bracy, D. Fan, D. Gries, L. Lee,
S. Marschner, C. Van Loan, W. White]

Announcements

- Download code from lecture and experiment with it—run, modify, run again, …
- Assignment 1 will be out around Friday
  - Will have over a week to do it
  - Can choose to work with one partner and together submit one assignment
  - Can revise and resubmit after getting grading feedback
- Starting next week: **optional 1-on-1** with a staff member to help *just you* with course material. Sign up for a slot on CMS under "SPECIAL: one-on-ones".
- Ed Discussions: you can post error msgs but do not post any amount of your code (answers)

Recall the Python API

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



math.ceil(x)
Return the ceiling of x, the smallest integer greater than or equal to x. If x is not a float, delegates to x.__ceil__(), which should return an Integral value.

9.2.1. Number-theoretic and repr

math.copysign(x, y)
Return a float with the magnitude (absolute value) … turns -1.0.

math.fabs(x)
Return the absolute value of x.

math.factorial(x)
Return x factorial. Raises ValueError if x is not integral or is negative.

- This is a **specification**
  - How to **use** the function
  - **Not** how to implement it
- Write them as **docstrings**

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

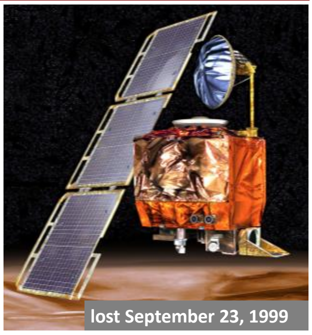
```
>>> get_campus_num("6072554444")
'5-4444'
>>> get_campus_num("6072531234")
'3-1234'
>>> get_campus_num(6072531234)
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
  File "/Users/Daisy/lec6examples.py", line 14, in get_campus_num
    return phone_num[5]+"-"+phone_num[6:10]
TypeError: 'int' object is not subscriptable
>>> get_campus_num("607-255-4444")
'5-5-44'
```

Precondition violated: **error message!**

Precondition violated: **no error message!**

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

10

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?

11

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!

12

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	More Test Cases
vowel_count('Bob') **Expect: 1**	vowel_count('y') **Expect: 0? 1?**
vowel_count('Aeiuo') **Expect: 5**	vowel_count('Bobo') **Expect: 1? 2?**
vowel_count('Grrr') **Expect: 0**	

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

13

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

14

Representative Tests Example

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

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

Look at precondition when choosing tests

Representative Tests:

- last_name_first('Katherine Johnson') Expects: 'Johnson, Katherine'
- last_name_first('Katherine   Johnson') Expects: 'Johnson, Katherine'

16

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 ?!

cornellasserts module

- Contains useful testing functions
- To use:
  - Download from course website (one of today's lecture files)
  - Put in same folder as the files you wish to test

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** cornellasserts **module** (supports testing)
  - **Defines one or more test cases** that each includes:
    - A representative input
    - The expected output
  - Test cases call a cornellasserts function:

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

Testing last_name_first(full_name)

```
import name_phone       # The module we want to test
import cornellasserts   # Module that supports testing

# First test case
result = name_phone.last_name_first('Katherine Johnson')
cornellasserts.assert_equals('Johnson, Katherine', result)

# Second test case
result = name_phone.last_name_first('Katherine      Johnson')
cornellasserts.assert_equals('Johnson, Katherine', result)

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

Actual output

Input

Expected output

Quits Python if actual and expected output not equal

Prints only if no errors

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 Case 1
    result = name.last_name_first('Katherine Johnson')
    cornellasserts.assert_equals('Johnson, Katherine', result)
    # Test Case 2
    result = name.last_name_first('Katherine      Johnson')
    cornellasserts.assert_equals('Johnson, Katherine', result)

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

Still need to import modules name_phone, cornellasserts

Put all test cases inside one function

No tests happen if you forget to call the function

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.index(' ')
       #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('Katherine Johnson') gives 'Johnson, Katherine'
- last_name_first('Katherine   Johnson') gives '   Johnson, Katherine'

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 if possible*)
2. Use print statements to reveal intermediate program states—variable values

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.index(' ')
    # 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("Katherine Johnson")
```

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

Using print statement to debug

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