Lecture 6: Specifications & Testing (Sections 4.9, 9.5)
CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2022sp

• We strongly encourage you to look at the `last_name_first` function in the Python tutor.
• Now try to fix the function implementation!

Welcome!

Please, no cell phones during lecture

Announcements
• 1-on-1s are happening and they are awesome!
  ▪ Sign up on CMS
• A1 goes out tonight! (many pages, but big figures)
• Academic Integrity Policy:
  ▪ You can talk to each other
  ▪ Do not show anyone (except staff) your code
  ▪ Do not post your code to Ed Discussions
  ▪ Do not look at anyone else's code
  ▪ The Full Policy: https://www.cs.cornell.edu/courses/cs1110/2022sp/policies/cs1110integrity.html

Asking Questions in Lecture
• Raise your hand for a notecard!
• Raise both hands for the catchBox!

Recall the Python API

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

• This is a specification
  ▪ How to use the function
  ▪ Not how to implement it
• Write them as docstrings
def greet(name):
    """Greets the person called 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?')

# Anatomy of a Specification (1)

# 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

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"

# A Precondition Is a Contract (1)

# If the precondition is met, the function will work!

campus.py

>>> import campus
>>> campus.get_campus_num("6072554444")
'5-4444'

# Software Bugs occur if

• Precondition is not documented properly
  § Easy to be unaware of assumptions we make

• Function use violates the precondition
  § Easy to think we're using a function properly, even if we're not

# A Precondition Is a Contract (2)

# If the precondition is not met...
# Sorry, no guarantees!

campus.py

>>> import campus
>>> campus.get_campus_num("6072554444")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/bracy/campus.py", line 7, in get_campus_num
TypeError: 'int' object is not subscriptable

# A Precondition Is a Contract (2)

# If the precondition is not met...
# Sorry, no guarantees!

campus.py

>>> import campus
>>> campus.get_campus_num("607-255-4444")
'5-5-44'

# If the precondition is not met...
# Sorry, no guarantees!

# 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...”

Preconditions Make Expectations Explicit

In American terms:
Preconditions help assign blame.
Something went wrong:
Engine breaks down.

Did you give the function a bad argument?
Did you put the wrong kind of fuel in the car?
OR
Was the function implemented/specified wrong?
Did the fuel tank ask for the wrong kind of fuel?
Was the engine simply poorly built?

Test cases help you find errors

def vowel_count(word):
    """Returns: number of vowels in word.
word: a string with at least one letter & 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?
- vowel_count('Bobo')
  Expect: 1?

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

Representative Tests Example

```python
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

Representative Tests:
>>> import name
>>> name.last_name_first('Katherine Jones')
Expects: 'Jones, Katherine'
>>> name.last_name_first('Katherine Jones')
Expects: 'Jones, Katherine'
```
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?!

Testing `last_name_first(full_name)`

- 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:

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

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

```python
import cornellasserts
import name
import campus

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

# Execution of the testing code
test_last_name_first()
print('All tests of the module name passed')
```
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
    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 Jones') gives 'Jones, Katherine'
last_name_first('Katherine    Jones') gives 'Jones, Katherine'

Which line is "wrong"?
A: Line 1
B: Line 2
C: Line 3
D: Line 4
E: I don't know

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):
    # 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

How do I print this?

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