

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

The image shows a screenshot of the Python documentation for the `math` module, specifically the `ceil` function. Several green callout boxes highlight key components of the API:

- Function name:** Points to `math.ceil(x)`.
- Possible arguments:** Points to the parameter `x` in `math.ceil(x)`.
- Module:** Points to the `math` module name in the function signature.
- What the function evaluates to:** Points to the description of the return value: "Return the ceiling of `x`, the smallest integer greater than or equal to `x`."

The background shows the Python Software Foundation documentation page for `9.2. math — Mathematical functions`. The page includes a table of contents, a search bar, and a list of functions provided by the module. The `ceil` function is described as follows:

Return the ceiling of `x`, the smallest integer greater than or equal to `x`.

The functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1 Arithmetic and representation functions

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.

math.copysign(x, y)
Return a float with the magnitude (absolute value) of `x` but the sign of `y`. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns `-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.

math.floor(x)
Return the floor of `x`, the largest integer less than or equal to `x`. If `x` is not a float, delegates to `x.__floor__()`, which should return an `Integral` value.

math.fmod(x, y)
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer `n` such that the result has the same sign as `x` and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of `y` instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be

Recall: The Python API

Function
name

Possible arguments

`math.ceil(x)`

Return the ceiling of `x`, the smallest integer greater than or equal to `x`.

Module

What the function evaluates to

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

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

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_centrigrade(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_centrigrade(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
 - `>>> to_centrigrade(32.0)`
0.0
 - If precondition is false, no guarantees at all
 - `>>> to_centrigrade(212)`
100.0
- Get **software bugs** when
 - Function precondition is not documented properly
 - Function is used in ways that violates precondition

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_centrigrade(32.0)
```

```
0.0
```

```
>>> to_centrigrade(212)
```

```
100.0
```

```
>>> to_centrigrade('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, run
- **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)
```

Quits Python
if not equal

```
# Second test case
```

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

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

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