

Lecture 5

# **Specifications & Testing**

# Announcements For This Lecture

---

## Readings

---

- See link on website:
  - Docstrings in Python
  - Material is not in Text

## Today's Lab

---

- Practice today's lecture
- **Preparation for the first Assignment**

## Assignment 1

---

- Posted on web page
  - Due Thu, Sep. 17<sup>th</sup>
  - Revise until correct
- Can work in pairs
  - One submission for pair
  - Meet people in your lab
- **Consultants can help**

# One-on-One Sessions

---

- Starting tomorrow: **1/2-hour one-on-one sessions**
  - Bring computer to work with instructor, TA or consultant
  - Hands on, dedicated help with Lab 2 and/or Lab 3
  - To prepare for assignment, **not for help on assignment**
- **Limited availability: we cannot get to everyone**
  - Students with experience or confidence should hold back
- Sign up online in CMS: first come, first served
  - Choose assignment One-on-One
  - Pick a time that works for you; will add slots as possible
  - Can sign up starting at 1pm **TODAY**

# Recall: The Python API

The image shows a screenshot of the Python documentation for the `math` module, specifically the `ceil` function. The page title is "9.2. math — Mathematical functions — Python v2.7.3 documentation". The function signature is `math.ceil(x)`. A callout box points to the function name `ceil`. Another callout box points to the parameter `x`. A third callout box points to the description: "Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`." A fourth callout box points to the return value description: "so that the programmer can determine how and why it was generated in the first place." The page also lists other functions in the module: `math.copysign(x, y)`, `math.fabs(x)`, `math.factorial(x)`, and `math.floor(x)`.

Function name

Number of arguments

What the function evaluates to

9.2. math — Mathematical functions — Python v2.7.3 documentation

python.org/library/math.html

python library

Entertainment (29) Commentarity (64) News (24) Research Puzzles Shopping Travel Financial Troubleshooting

Table Of Contents 9.2. math — Mathematical functions

previous | next | modules | index

`math.ceil(x)`

Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all

representation functions

9.1. numbers — Numeric abstract base classes

Next topic

9.3. cmath — Mathematical functions for complex numbers

This Page

Report a Bug

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

`math.ceil(x)`

Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

`math.copysign(x, y)`

Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

*New in version 2.6.*

`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.

*New in version 2.6.*

`math.floor(x)`

Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

# Recall: The Python API

The screenshot shows the Python documentation for `math.ceil(x)`. A green callout bubble points to `ceil` with the text "Function name". Another green callout bubble points to `(x)` with the text "Number of arguments". A third green callout bubble points to the description "Return the ceiling of x as a float, the smallest integer value greater than or equal to x." with the text "What the function evaluates to".

9.2. math — Mathematical functions — Python v2.7.3 documentation

python.org/library/math.html

Entertainment (29) | Commentary (64) | News (24) | Research | Puzzles | Shopping | Travel | Financial | TroubleShooting

Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » previous | next | modules | index

Table Of Contents | 9.2. math — Mathematical functions

`math.ceil(x)`

Return the ceiling of  $x$  as a float, the smallest integer value greater than or equal to  $x$ .

so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all

representation functions

9.1. numbers — Numeric abstract base classes

Next topic

9.3. cmath — Mathematical functions for complex numbers

This Page

Report a Bug

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

`math.ceil(x)`  
Return the ceiling of  $x$  as a float, the smallest integer value greater than or equal to  $x$ .

`math.copysign(x, y)`  
Return  $x$  with the sign of  $y$ . On a platform where  $float$  has IEEE 754 floating point arithmetic, the sign of  $0$  is always positive. *New in version 2.6.*

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

`math.factorial(x)`  
Return  $x$  factorial. Raises `ValueError` if  $x$  is not a non-negative integer. *New in version 2.6.*

`math.floor(x)`  
Return the floor of  $x$  as a float, the largest integer value less than or equal to  $x$ .

- 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_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
```

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
  - 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
```

```
>>> to_centrigrade(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
```

```
>>> 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

# Global Variables and Specifications

---

- Python *does not support* docstrings for variables
  - Only functions and modules (e.g. first docstring)
  - `help()` shows “data”, but does not describe it
- But we still need to document them
  - Use a single line comment with `#`
  - Describe what the variable means
- **Example:**
  - `FREEZING_C = 0.0 # temp. water freezes in C`
  - `BOILING_C = 100.0 # temp. water boils in C`

# 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

# 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 Module

---

- A unit test is a module that tests another module
  - It **imports the other module** (so it can access it)
  - It **imports the `cornelltest` module** (for testing)
  - It **defines one or more test procedures**
    - Evaluate the function(s) on the test cases
    - Compare the result to the expected value
  - It has special code that **calls the test procedures**
- The test procedures use the `cornelltest` function

```
def assert_equals(expected,received):
```

```
    """Quit program if expected and received differ"""
```

# Testing last\_name\_first(n)

```
# test procedure
```

```
def test_last_name_first():
```

```
    """Test procedure for last_name_first(n)"""
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

Quits Python  
if not equal

```
# Execution of the testing code
```

```
test_last_name_first()
```

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

Message will print  
out only if no errors.

# Testing last\_name\_first(n)

```
# test procedure
```

```
def test_last_name_first():
```

```
    """Test procedure for last_name_first(n)"""
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

Call function  
on test input

Quits Python  
if not equal

```
# Execution of the testing code
```

```
test_last_name_first()
```

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

Message will print  
out only if no errors.

# Testing last\_name\_first(n)

```
# test procedure
```

```
def test_last_name_first():
```

```
    """Test procedure for last_name_first(n)"""
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

Call function  
on test input

Compare to  
expected output

Quits Python  
if not equal

```
# Execution of the testing code
```

```
test_last_name_first()
```

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

Message will print  
out only if no errors.

# Modules vs. Scripts

---

## Module

---

- Provides functions, constants
  - **Example:** temp.py
- import it into Python
  - In interactive shell...
  - or other module
- All code is either
  - In a function definition, or
  - A variable assignment

## Script

---

- Behaves like an application
  - **Example:** helloApp.py
- Run it from command line
  - python helloApp.y
  - No interactive shell
  - import acts “weird”
- Commands *outside* functions
  - Does each one in order

# Combining Modules and Scripts

---

- Scripts often have functions in them
  - Can we import them without “running” script?
  - Want to separate script part from module part
- New feature: **if** `__name__ == '__main__':`
  - Put all “script code” underneath this line
  - Also, indent all the code underneath
  - Prevents code from running if imported
  - **Example:** `bettertemp.py`

# Testing last\_name\_first(n)

---

```
# test procedure
```

```
def test_last_name_first():
```

```
    """Test procedure for last_name_first(n)"""
```

```
    result = name.last_name_first('Walker White')
```

```
    cornelltest.assert_equals('White, Walker', result)
```

```
    result = name.last_name_first('Walker      White')
```

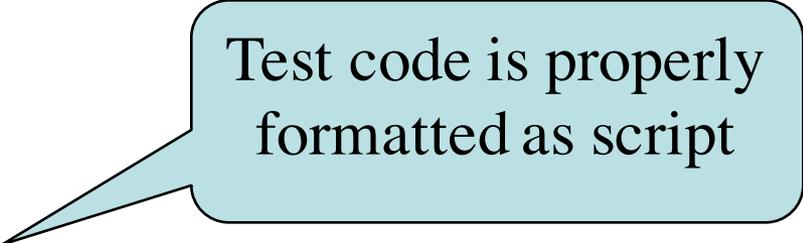
```
    cornelltest.assert_equals('White, Walker', result)
```

```
# Application code
```

```
if __name__ == '__main__':
```

```
    test_last_name_first()
```

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



Test code is properly formatted as script

# Modules/Scripts in this Course

- Our modules consist of
  - Function definitions
  - “Constants” (global vars)
  - **Optional** script code to call/test the functions
- All **statements** must
  - be inside of a function **or**
  - assign a constant **or**
  - be in the application code
- import will only use the definitions, not app code

```
# temperature.py
...
# Functions
def to_centigrade(x):
    | """Returns: x converted to C"""
...
# Constants
FREEZING_C = 0.0 # temp. water freezes
...
# Application code
if __name__ == '__main__':
    | assert_floats_equal(0.0,to_centigrade(32.0))
    | assert_floats_equal(100,to_centigrade(212))
    | assert_floats_equal(32.0,to_fahrenheit(0.0))
    | assert_floats_equal(212.0,to_fahrenheit(100.0))
```

# Types of Testing

---

## Black Box Testing

---

- Function is “opaque”
  - Test looks at what it does
  - **Fruitful**: what it returns
  - **Procedure**: what changes
- **Example**: Unit tests
- **Problems**:
  - Are the tests everything?
  - What caused the error?

## White Box Testing

---

- Function is “transparent”
  - Tests/debugging takes place inside of function
  - Focuses on where error is
- **Example**: Use of print
- **Problems**:
  - Much harder to do
  - Must remove when done

# Finding the Error

- Unit tests cannot find the source of an error
- Idea: “Visualize” the program with print statements

```
def last_name_first(n):
```

```
    """Returns: copy of <n> in form <last>, <first>"""
```

```
    end_first = n.find(' ')
```

```
    print end_first
```

```
    first = n[:end_first]
```

```
    print 'first is '+str(first)
```

```
    last = n[end_first+1:]
```

```
    print 'last is '+str(last)
```

```
    return last+', '+first
```

Print variable after each assignment

**Optional:** Annotate value to make it easier to identify