

Lecture 6

Specifications & Testing

Announcements For This Lecture

Last Call

- Acad. Integrity Quiz
- Take it by tomorrow
- Also remember survey



Assignment 1

- Posted on web page
 - Due Fri, Sep. 16th
 - Today's lab will help
 - Revise until correct
- Can work in pairs
 - We will pair if needed
 - Submit request Friday
 - One submission per pair

One-on-One Sessions

- Starts Sunday: 1/2-hour one-on-one sessions
 - Bring computer to work with instructor, TA or consultant
 - Hands on, dedicated help with Labs 6 (and related)
 - 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 5pm **TOMORROW**

Recall: The Python API

The image shows a screenshot of the Python documentation for the `math.ceil(x)` function. Several green callout boxes highlight key parts of the documentation:

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

The documentation text includes:

9.2. `math` — Mathematical functions

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

float, delegates to `x._ceil_()`, which should return an

Previous topic
9.1. `numbers` — Numeric abstract base classes

Next topic
9.3. `cmath` — Mathematical functions for complex numbers

This Page
Report a Bug
Show Source

`math.copysign(x, y)`
Return a float with the magnitude (absolute value) of `y` and the sign of `x`.
-1.0.

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

`math.factorial(x)`
Return `x` factorial. Raises `ValueError` if `x` is not integer.

`math.floor(x)`
Return the floor of `x`, the largest integer less than or equal to `x`.

`math.fmod(x, y)`
Return `fmod(x, y)`, as defined by the platform C library. The C standard is that `fmod(x, y)` be exactly (mathematically) the same sign as `x` and magnitude less than `abs(y)`. Python 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

- This is a **specification**
 - Enough info to **call** function
 - But not how to **implement**
- Write them as **docstrings**

Anatomy of a Specification

```
def greet(n):
```

One line description,
followed by blank line

```
    """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?')
```

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.

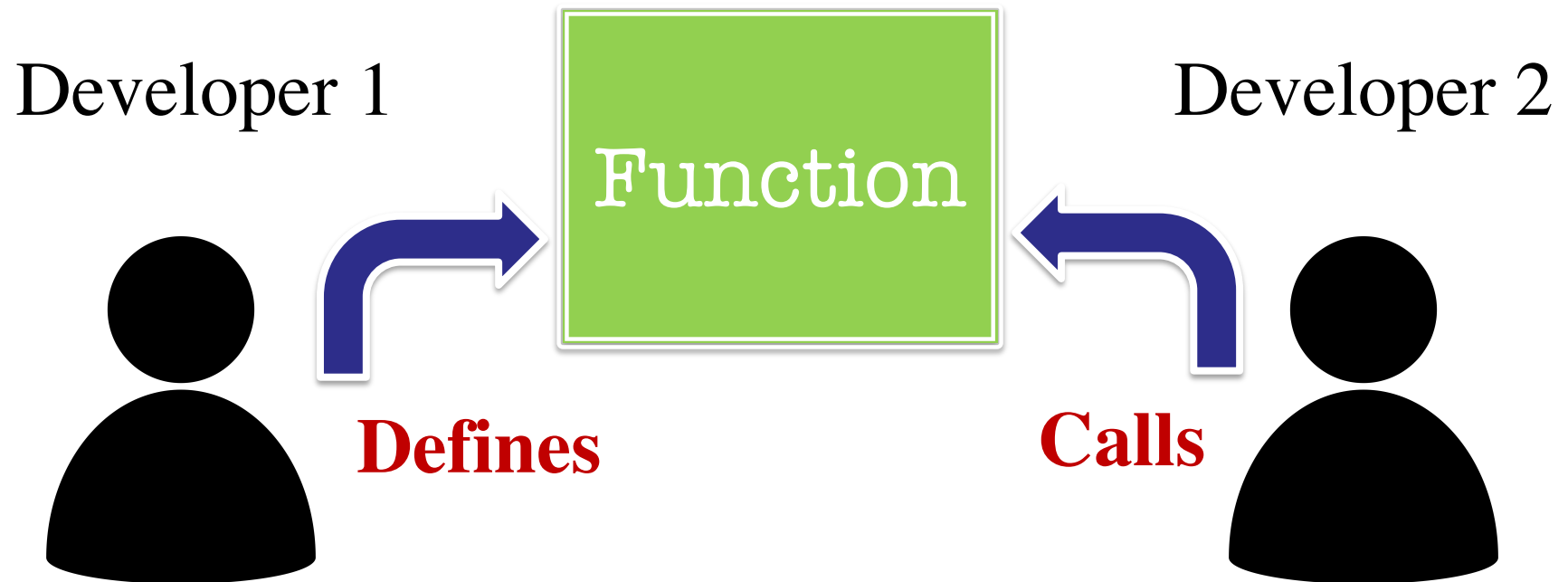
Parameter description

Precondition specifies assumptions we make about the arguments

What Makes a Specification “Good”?

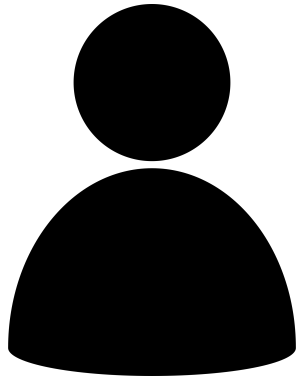
- Software development is a **business**
 - Not just about coding – **business processes**
 - Processes enable better code development
- Complex projects need **multi-person** teams
 - Lone programmers do simple contract work
 - Teams must have people working separately
- Processes are about how to **break-up** the work
 - What pieces to give each team member?
 - How can we fit these pieces back together?

Functions as a Way to Separate Work



Working on Complicated Software

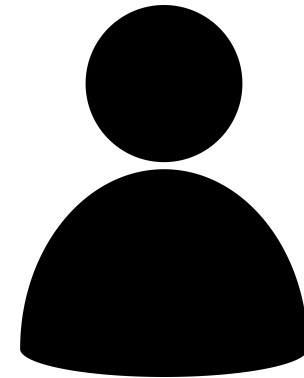
Developer 1



Calls



Developer 2



Func 1

Func 2

Func 4

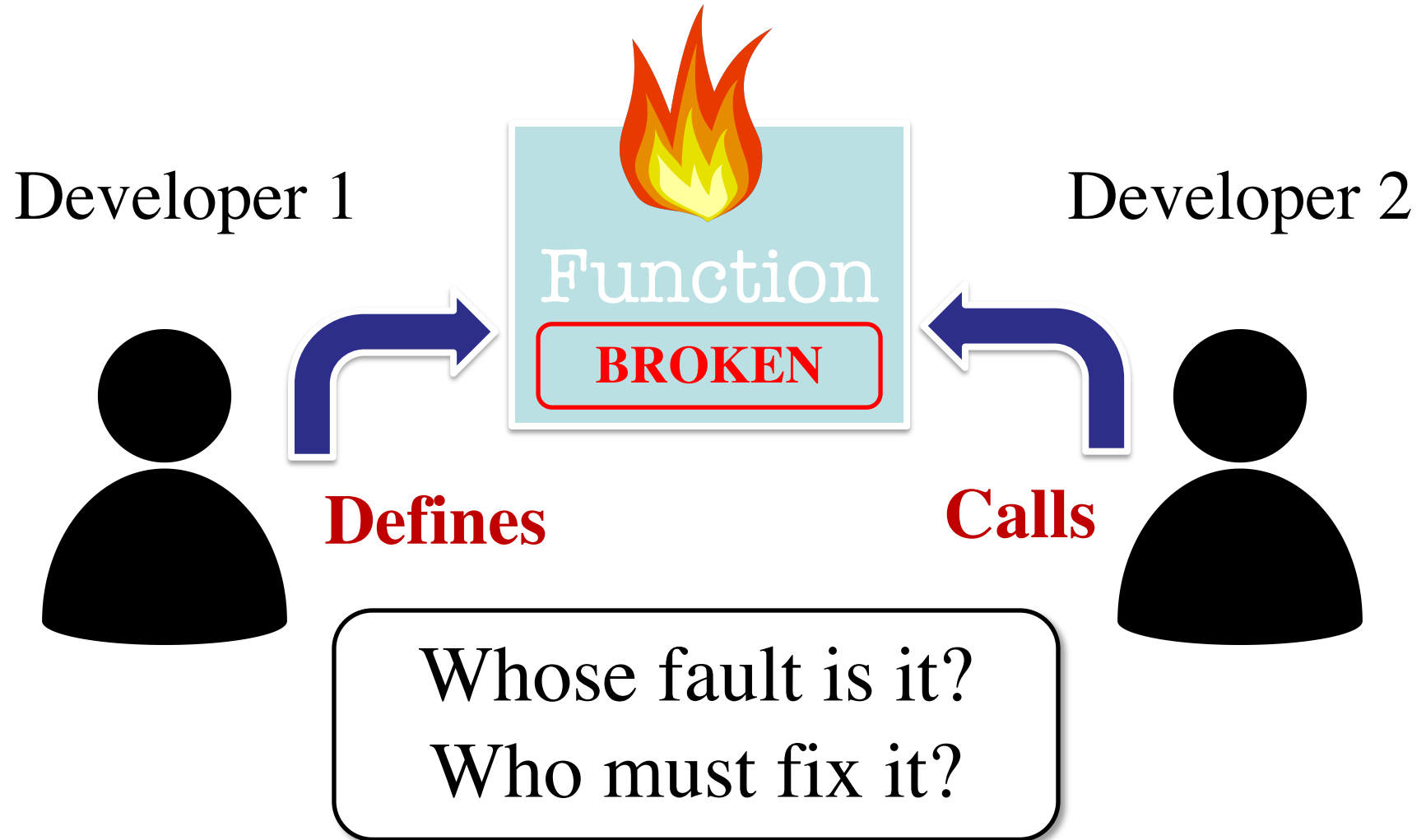
Func 5

Func 3

Architect plans
the separation

Func 6

What Happens When Code Breaks?



Purpose of a Specification

- To clearly layout **responsibility**
 - What does the function promise to do?
 - What is the allowable use of the function?
- From this responsibility we determine
 - If definer implemented function properly
 - If caller uses the function in a way allowed
- A specification is a **business contract**
 - Requires a formal documentation style
 - Rules for modifying contract *beyond course scope*

Preconditions are a Promise

- If precondition true
 - Function must work
- If precondition false
 - Function might work
 - Function might not
- Assigns responsibility
 - How to tell fault?

```
>>> to_centrigrade(32.0)
```

```
0.0
```

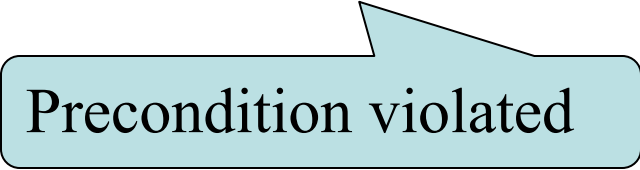
```
>>> to_centrigrade('32')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

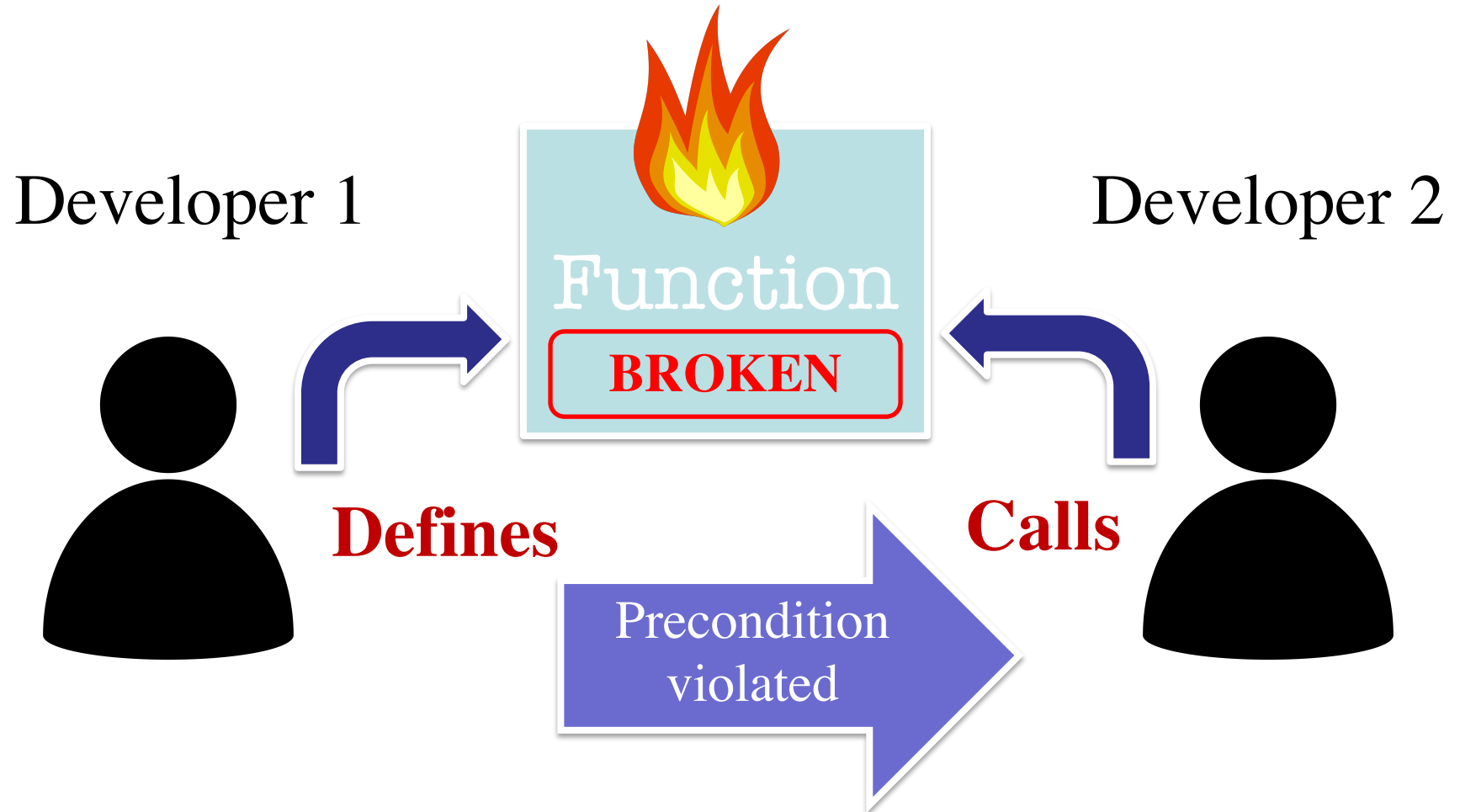
```
File "temperature.py", line 19 ...
```

```
TypeError: unsupported operand type(s)  
for -: 'str' and 'int'
```

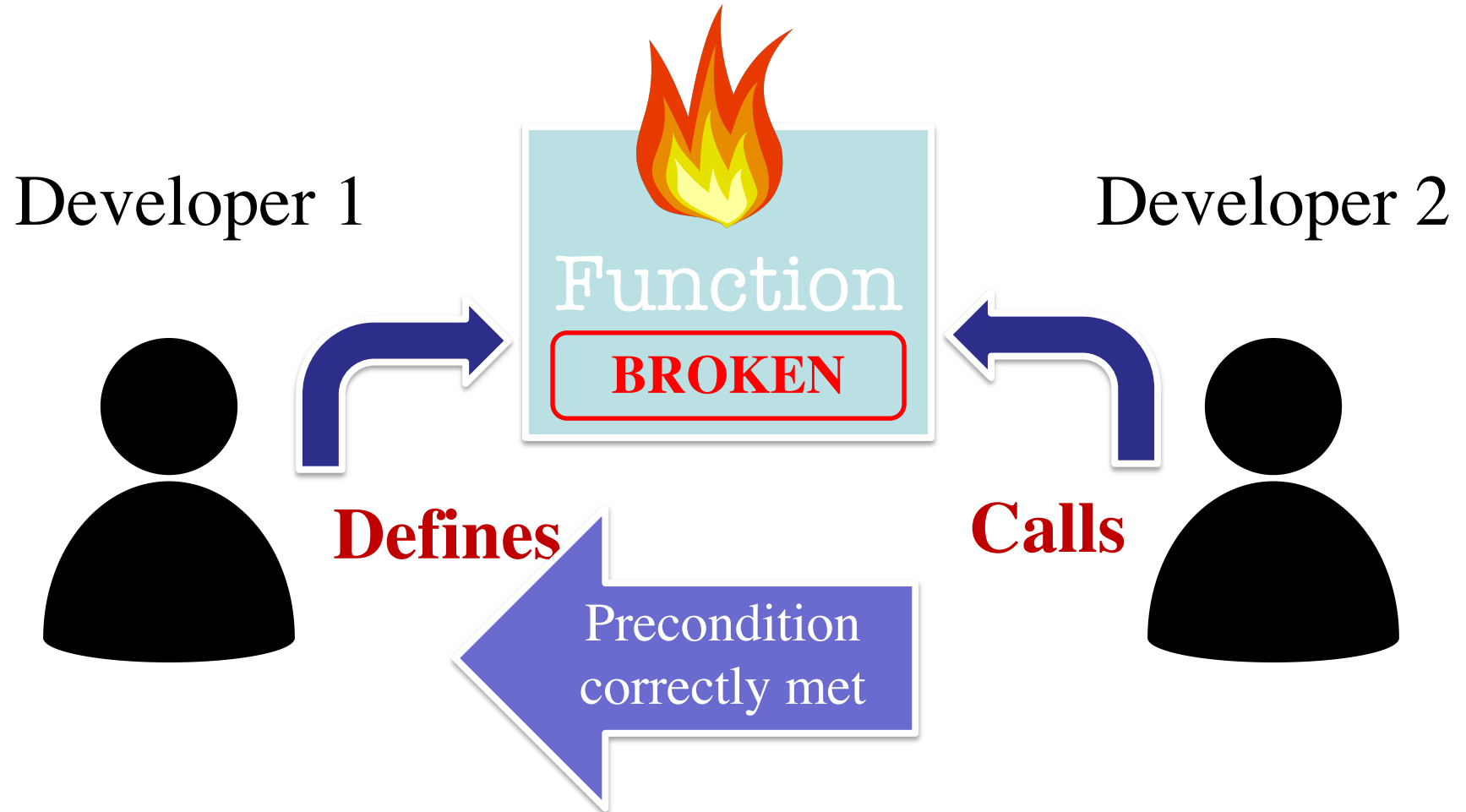


Precondition violated

Assigning Responsibility



Assigning Responsibility



What if it Just Works?

- Violation != crash
 - Sometimes works anyway
 - *Undocumented* behavior
- But is **bad practice**
 - Definer may change the definition at any time
 - Can do anything so long as specification met
 - Caller code breaks
- Hits Microsoft devs a lot

```
>>> to_centrigrade(32.0)
```

```
0.0
```

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

```
100.0
```

Precondition violated

Precondition
violations are
unspecified!

Testing Software

- You are **responsible** for your function definition
 - You must ensure it meets the specification
 - May even need to prove it to your boss
- **Testing**: Analyzing & running a program
 - Part of, but not the same as, **debugging**
 - Finds **bugs** (errors), but does not remove them
- To test your function, you create a **test plan**
 - A test plan is made up of several **test cases**
 - Each is an **input** (argument), and its expected **output**

Test Plan: A Case Study

```
def number_vowels(w):
```

```
    """
```

```
    Returns: number of vowels in string w.
```

```
    Parameter w: The text to check for vowels
```

```
    Precondition: w string w/ at least one letter and only letters
```

```
    """
```

```
    ...
```

Brainstorm
some test cases

Test Plan: A Case Study

```
def number_vowels(w):
```

```
    """
```

```
    Returns: number of vowels in string w.
```

```
    Parameter w: The text to check for vowels
```

```
    Precondition: w string w/ at least one letter and only letters
```

```
    """
```

```
    ...
```

rhythm?
crwth?

Surprise!
Bad Specification

Test Plan: A Case Study

```
def number_vowels(w):
```

```
    """
```

```
    Returns: number of vowels in string w.
```

```
    Vowels are defined to be 'a','e','i','o', and 'u'. 'y' is a vowel if it is  
    not at the start of the word.
```

```
    Repeated vowels are counted separately. Both upper case and  
    lower case vowels are counted.
```

```
    Examples: ....
```

```
    Parameter w: The text to check for vowels
```

```
    Precondition: w string w/ at least one letter and only letters
```

```
    """
```

Test Plan: A Case Study

```
def number_vowels(w):
```

```
    """
```

```
    Returns: number of vowels
```

```
    Vowels are defined to be 'a',  
    not at the start of the word
```

```
    Repeated vowels are counted separately. Both upper case and  
    lower case vowels are counted.
```

```
    Examples: ....
```

```
    Parameter w: The text to check for vowels
```

```
    Precondition: w string w/ at least one letter and only letters
```

```
    """
```

Some Test Cases

INPUT	OUTPUT
'hat'	1
'aeiou'	5
'grrr'	0

Representative Tests

- We cannot test all possible inputs
 - “Infinite” possibilities (strings arbitrary length)
 - Even if finite, way too many to test
- Limit to tests that are **representative**
 - Each test is a significantly different input
 - Every possible input is similar to one chosen
- This is an **art**, not a **science**
 - If easy, no one would ever have bugs
 - Learn with much practice (and why teach early)

Representative Tests

Representative Tests for `number_vowels(w)`

Simplest
case first!

A little
complex

“Weird”
cases

- 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

The Rule of Numbers

- When testing the numbers are 1, 2, and 0
- **Number 1**: The simplest test possible
 - If a complex test fails, what was the problem?
 - **Example**: Word with just one vowels
- **Number 2**: Add more than was expected
 - **Example**: Multiple vowels (all ways)
- **Number 0**: Make something missing
 - **Example**: Words with no vowels

Running Example

- The following function has a bug:

```
def last_name_first(n):  
    """Returns a copy of n in the form 'last-name, first-name'  
  
    Precondition: n is in the form 'first-name last-name'  
    with one or more spaces between the two names"""  
    end_first = n.find(' ')  
    first = n[:end_first]  
    last = n[end_first+1:]  
    return last+', '+first
```

Precondition
forbids a 0th test

- Representative Tests:
 - `last_name_first('Walker White')` returns 'White, Walker'
 - `last_name_first('Walker     White')` returns 'White, Walker'

Test Scripts: Automating Testing

- To test a function we have to 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 file?
 - This file is a **script** to test the **module**

Unit Test: An Automated Test Script

- A **unit test** is a script to test a **single function**
 - Imports the function module (so it can access it)
 - Imports the **intprocs** module (for testing)
 - Implements one or more test cases
 - A representative input
 - The expected output
- The test cases use the **intprocs** 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 intros              # Includes the test procedures

# Test one space between names
result = name.last_name_first('Walker White')
intros.assert_equals('White, Walker', result)

# Test multiple spaces between names
result = name.last_name_first('Walker      White')
intros.assert_equals('White, Walker', result)

print('Module name passed all tests.')
```

Testing last_name_first(n)

```
import name                # The module we want to test
import intros              # Include
```

Comment
describing test

```
# Test one space between names
result = name.last_name_first('Walker White')
intros.assert_equals('White, Walker', result)
```

Actual Output

```
# Test multiple spaces between names
result = name.last_name_first('Walker
                               White')
intros.assert_equals('White, Walker', result)
```

Input

Expected Output

```
print('Module name passed all tests.')
```

Testing last_name_first(n)

```
import name                # The module we want to test
import intros              # Includes the test procedures
```

```
# Test one space between names
```

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

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

Quits Python
if not equal

```
# Test multiple spaces between names
```

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

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

```
print('Module name passed all tests.')
```

Message will print
out only if no errors.

Testing Multiple Functions

- Unit test is for a **single function**
 - But you are often testing many functions
 - Do not want to write a test script for each
- **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')  
    introcs.assert_equals('White, Walker', result)  
    result = name.last_name_first('Walker      White')  
    introcs.assert_equals('White, Walker', result)  
  
# Execution of the testing code  
test_last_name_first()  
print('Module name passed all tests.')
```

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

Execution of the testing code

test_last_name_first()

print('Module name passed all tests.')

No tests happen
if you forget this