Module 13

Errors and Asserts

Motivation

- Specifications assign responsibility
  - When code crashes, who is responsible?
- But this is not always immediately clear
  - Have to read & interpret specification
  - Must compare with what actually happened
- Need to understand error messages
  - Error messages tell us what happened
  - But there is a lot of “hidden” detail

Error Messages

Not An Error Message

```
ZeroDivisionError: division by zero
```

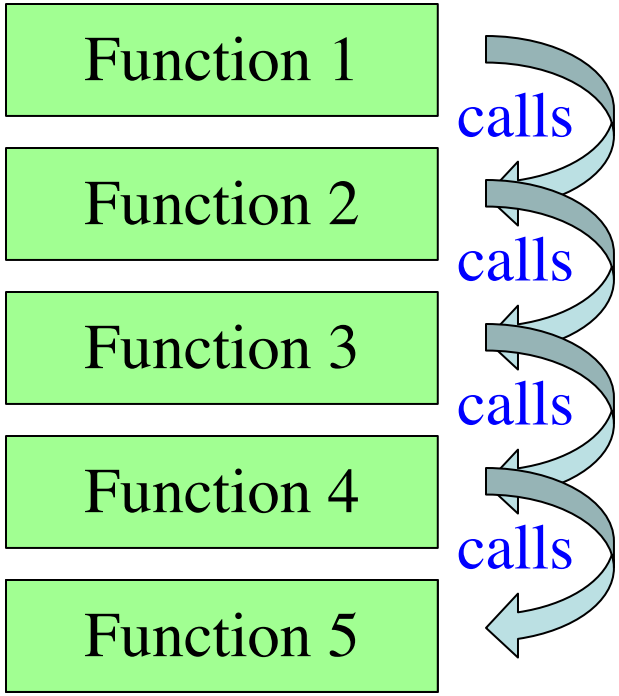
Everything starting with the Traceback

An Error Message

```
Traceback (most recent call last):
  File "error.py", line 41, in <module>
    print(function_1(1,0))
  File "error.py", line 16, in function_1
    return function_2(x,y)
  File "error.py", line 26, in function_2
    return function_3(x,y)
  File "error.py", line 36, in function_3
    return x/y
ZeroDivisionError: division by zero
```

Recall: The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
  - Top down because of Tutor
- Effects your memory
  - Need RAM for **entire stack**
  - An issue in adv. programs



Errors and the Call Stack

```
# error.py

def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here

if __name__ == '__main__':
    print(function_1(1,0))
```

Crashes produce the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print(function_1(1,0))
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y
```

Errors and the Call Stack

```
#

d
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here

if
```

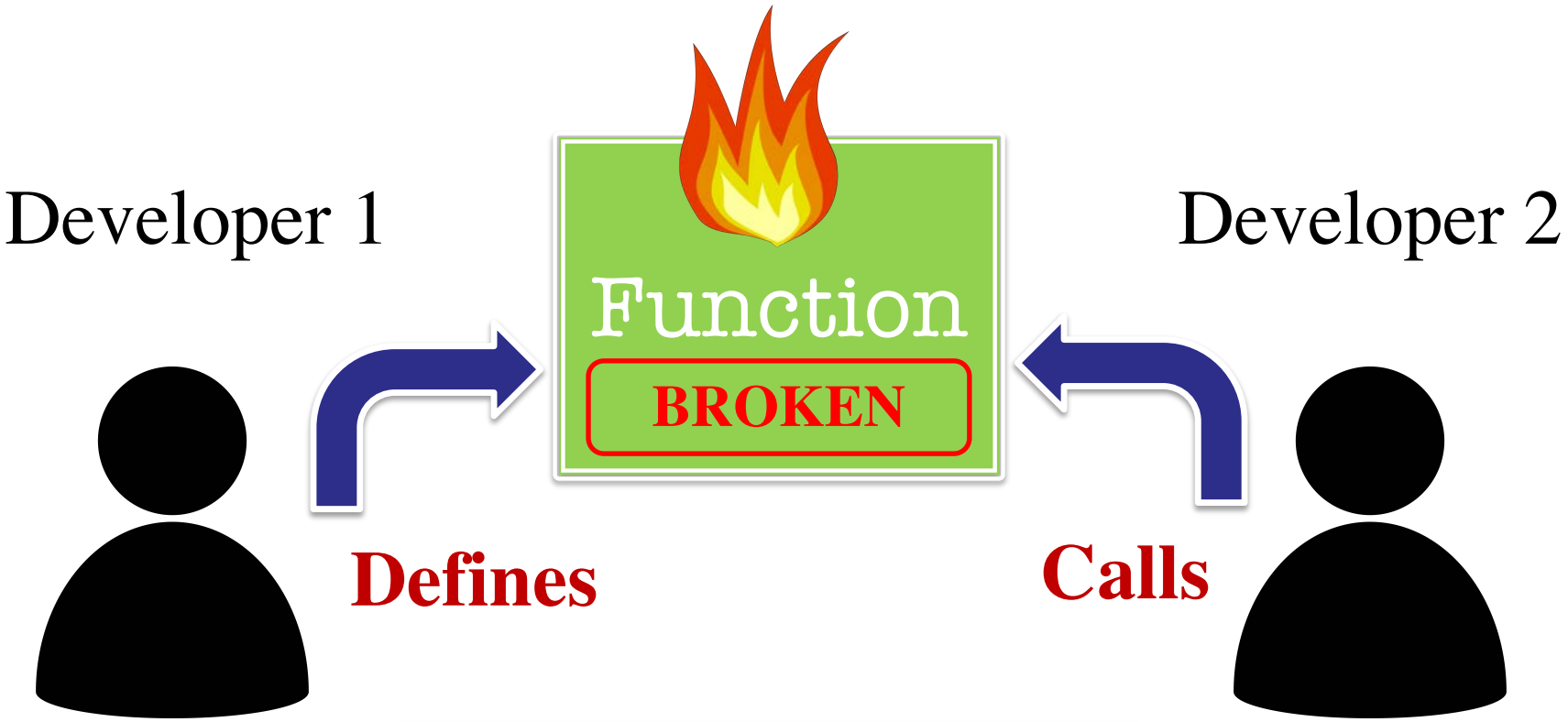
Script code.
Global space

Where error occurred
(or where was found)

Crashes produce the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print(function_1(1,0))
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y
```

Recall: Assigning Responsibility



Whose fault is it?
Who must fix it?

Determining Responsbility

```
def function_1(x,y):
    """Returns: result of function_2

    Precondition: x, y numbers"""
    return function_2(x,y)

def function_2(x,y):
    """Returns: x divided by y

    Precondition: x, y numbers"""
    return x/y

print(function_1(1,0))
```

```
Traceback (most recent call last):

  File "error1.py", line 32, in <module>
    print(function_1(1,0))

  File "error1.py", line 18, in function_1
    return function_2(x,y)

  File "error1.py", line 28, in function_2
    return x/y

ZeroDivisionE
```

Where is the error?

Approaching the Error Message

- Start from the top
- Look at function call
  - Examine arguments
  - (Print if you have to)
  - Verify preconditions
- Violation? Error found
  - Else go to next call
  - Continue until bottom

```
Traceback (most recent call last):

  File "error1.py", line 32, in <module>
    print(function_1(1,0))

  File "error1.py", line 18, in function_1
    return function_2(x,y)

  File "error1.py", line 28, in function_2
    return x/y

ZeroDivisionError: division by zero
```

Determining Responsbility

```
def function_1(x,y):
    """Returns: result of function_2

    Precondition: x, y numbers"""
    return function_2(x,y)

def function_2(x,y):
    """Returns: x divided by y

    Precondition: x, y numbers"""
    return x/y

print(function_1(1,0))
```

```
Traceback (most recent call last):

  File "error1.py", line 32, in <module>
    print(function_1(1,0))

  File "error1.py", line 18, in function_1
    return function_2(x,y)

  File "error1.py", line 28, in function_2
    return x/y

ZeroDivisionError: division by zero
```

Error!

Determining Responsbility

```
def function_1(x,y):
    """Returns: result of function_2

    Precondition: x, y numbers"""
    return function_2(x,y)


def function_2(x,y):
    """Returns: x divided by y

    Precondition: x, y numbs, y > 0"""
    return x/y


print(function_1(1,0))
```

```
Traceback (most recent call last):

  File "error1.py", line 32, in <module>
    print(function_1(1,0))

  File "error1.py", line 18, in function_1
    return function_2(x,y)

  File "error1.py", line 28, in function_2
    return x/y

ZeroDivisionError: division by zero
```

Error!

Determining Responsbility

```
def function_1(x,y):
    """Returns: result of function_2

    Precondition: x, y numbs, y > 0"""
    return function_2(x,y)

def function_2(x,y):
    """Returns: x divided by y

    Precondition: x, y numbs, y > 0"""
    return x/y

print(function_1(1,0))
```

```
Traceback (most recent call last):

  File "error1.py", line 32, in <module>
    print(function_1(1,0))

  File "error1.py", line 18, in function_1
    return function_2(x,y)

  File "error1.py", line 28, in function_2
    return x/y

ZeroDivisionError: division by zero
```

Error!

Aiding the Search Process

- We talked about assigning responsibility
  - Have to step through the error message
  - Compare to specification at each step
- How can we make this easier?
  - What if we could control the error messages
  - Write responsibility directly into error
  - Then only need to look at error message
- We do this with **assert statements**

Assert Statements

- **Form 1:** `assert` <boolean>
  - Does nothing if boolean is True
  - Creates an error is boolean is False
- **Form 2:** `assert` <boolean>, <string>
  - Very much like form 2
  - But error message includes the message
- Statement to verify a fact is true
  - Similar to `assert_equals` used in unit tests
  - But more versatile with complete stack trace

Enforcing Preconditions

- **Idea:** Assert all of the preconditions
  - If preconditions violated, crash immediately
  - Message immediately indicates the problem
- Error position is now immediately clear
  - Error was the call to this function
  - Occurs in line BEFORE in the stack trace
- **Example:** last_name_first

Enforcing Preconditions

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    n has only space, separating first and last."""
    assert type(n) == str, 'Precondition violation'
    assert count_str(n,' ') == 1, 'Precondition violation'
    # Implement method here...
```

Another Advantage

- Undocumented behavior now impossible
  - ALL violations guaranteed to crash
  - Only valid calls execute normally
- Generally considered a good thing
  - Undocumented behavior can metastasize
  - Shuts it down before it can get any worse
- **Example:** to_centigrade(x)

Eliminating Undocumented Behavior

```
def to_centigrade(x):
    """Returns: x converted to centigrade

    Parameter x: temp in fahrenheit
    Precondition: x is a float"""
    assert type(n) == float, 'Precondition violation'
    # Implement method here...
```

Will do yourself in **A4**.

Recall: Enforcing Preconditions

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    n has only space, separating first and last."""
    assert type(n) == str,          'Precondition violation'
    assert count_str(n,' ') == 1,   'Precondition violation'
    # Implement method here...
```

Can we do better?

Making Better Error Messages

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    n has only space, separating first and last."""
    assert type(n) == str,          str(n)+' is not a string'
    assert count_str(n,' ') == 1,   n+' has the wrong form'
    # Implement method here...
```

We know n is a string

The Problem With Error Messages

```
>>> msg = str(var)+' is invalid'
>>> print(msg)
2 is invalid
```

- Looking at this output, what is the type of var?

A: **int**
B: **float**
C: **str**
D: Impossible to tell

The Problem With Error Messages

```
>>> msg = str(var)+' is invalid'
>>> print(msg)
2 is invalid
```

- Looking at this output, what is the type of var?

A: **int**
B: **float**
C: **str**
D: Impossible to tell

CORRECT

The Problem With Error Messages

```
>>> var = 2
>>> msg = str(var)+' is invalid'
>>> print(msg)
2 is invalid
>>> var = '2'
>>> msg = str(var)+' is invalid'
>>> print(msg)
2 is invalid
```

The Function repr

- Like str(), turns any value into a string
  - Built-in function provided by Python
  - Useful for concatentating value to string
- But formatted to represent original type
  - str('2') returns '2'
  - repr('2') returns "'2'" (includes quotes)
- Stands for “representation”

Error Messages with repr

```
>>> msg = str(var)+' is invalid'
>>> print(msg)
2 is invalid


>>> msg = repr(var)+' is invalid'
>>> print(msg)
'2' is invalid
```

Clear that var
is really a string

Enforcing Preconditions is Tricky!

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    There is one or more spaces separating first and last.
    There is no space in either the first or last name"""
    assert ?????
```

Assert use expressions only.
Each one must fit on one line.

This is an advanced precondition

Asserts are Never Required

- Some preconditions are **hard to express**
- Sometimes it is **too expensive**
  - Checking the precondition takes time
  - Sometimes you want the code to run fast
  - Why have asserts if confident no bugs
- In the end, only the specification matters
  - Asserts were there as a convenience
  - Used to help assign responsibility

How About a Compromise?

- Break precondition up into several parts
  - Sometimes this is clear from the specification
- **Assert** the things that are **easy** to check
  - This gives us some minimal enforcement
  - Allows us to identify the biggest errors
- **Omit** the things that are **hard** to check
  - Will just let that behavior go unchecked
  - Will catch it in the system some other way

Picking a Compromise

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    There is one or more spaces separating first and last.
    There is no space in either the first or last name"""
    assert type(n) == str     # Check the type
    assert ' ' in n           # Least we can say of space
    # Do not try to enforce anything else
```

Enforcing Preconditions is Tricky!

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    There is one or more spaces separating first and last.
    There is no space in either the first or last name"""
    assert ?????
```

Assert use expressions only.
Each one must fit on one line.

This is an advanced precondition

A Useful Function

```
def is_two_words(w):
    """Returns: True if w is 2 words sep by 1 or more spaces.

    A word is a string with no spaces.  So this means that
    1. The first characters is not a space (or empty)
    2. The last character is not a space (or empty)
    3. There is at least one space in the middle
    4. If there is more than one space, the spaces are adjacent
    Precondition: w is a str"""

    # implement me
```

A Useful Function

```
def is_two_words(w):
    """Returns: True if w is 2 words sep by 1 or more spaces.
    Precondition: w is a str"""
    if not ' ' in w:
        return False

    first = w.find(' '); last = w.rfind(' ')

    w0 = w[:first]; w2 = w[last+1:]
    w1 = w[first:last+1]

    cond1 = w1.count(' ') == len(w1)
    cond0 = w0 != ''; cond2 = w2 != ''

    return cond0 and cond1 and cond2
```

Find spaces

Cut in 3 parts

Check parts ok

Enforcing with The Second Function

```
def last_name_first(n):
    """Returns: copy of n in form 'last-name, first-name'
    Precondition: n string in form 'first-name last-name
    There is one or more spaces separating first and last.
    There is no space in either the first or last name"""
    assert type(n) == str
    assert is_two_words(n)
```

Rules for Using Helpers

- The function must return a Boolean
  - True/False and no other options
- It CAN have its own preconditions
  - But should be things checked so far
  - **Example:** n is a string
- Often does not enforce own preconditions
  - Only used by you (definer and caller)
  - Would just be redundant

A Useful Function

```
def is_two_words(w):
    """Returns: True if w is 2 words sep by 1 or more spaces.
    Precondition: w is a str"""
    if not ' ' in w:
        return False

    first = w.find(' '); last = w.rfind(' ')

    w0 = w[:first]; w2 = w[last+1:]
    w1 = w[first:last+1]

    cond1 = w1.count(' ') == len(w1)
    cond0 = w0 != '';  cond2 = w2 != ''

    return cond0 and cond1 and cond2
```

Precondition not enforced