Lecture 11

Asserts and Error Handling

Announcements for Today

Reading

- Reread Chapter 3
- 10.0-10.2, 10.4-10.6 for Tue

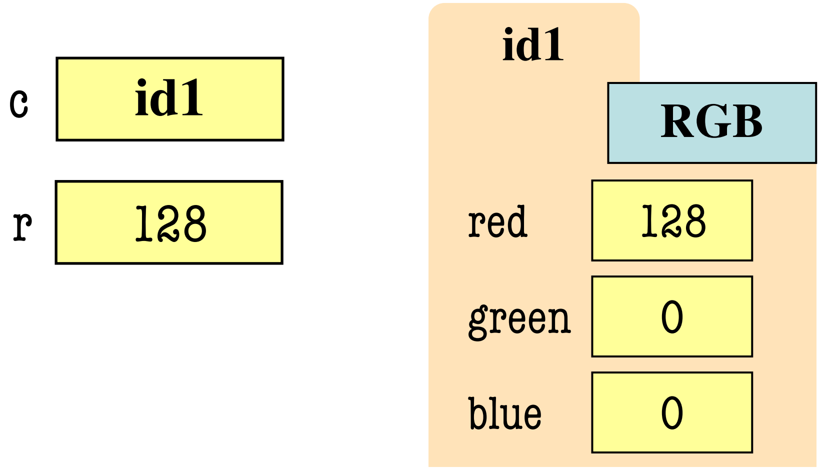
- **Prelim, Oct 17th 7:30-9:00**
  - Material up October 8th
  - Study guide next week
- **Conflict with Prelim time?**
  - Submit to Prelim 1 Conflict assignment on CMS
  - Do not submit if no conflict

Assignments

- Finishing **Assignment 1**
  - We are going to score it
  - Get **one more** chance Sun.
- **Assignment 2** in progress
  - Will grade it by Friday
  - Solutions posted by Friday
- **Assignment 3** due next week
  - Before you leave for break
  - Same “length” as A1

Using Color Objects in A3

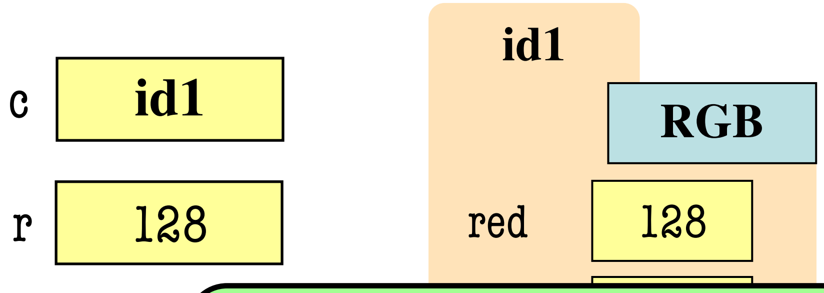
- New classes in introcs
  - RGB, CMYK, and HSV
- Each has its own attributes
  - **RGB**: red, blue, green
  - **CMYK**: cyan, magenta, yellow, black
  - **HSV**: hue, saturation, value
- Attributes have *invariants*
  - Limits the attribute values
  - Example: red is int in 0..255
  - Get an error if you violate



```
>>> import introcs
>>> c = introcs.RGB(128,0,0)
>>> r = c.red
>>> c.red = 500 # out of range
AssertionError: 500 outside [0,255]
```

Using Color Objects in A3

- New classes in introcs
  - RGB, CMYK, and HSV
- Each has its own attributes
  - **RGB**: red, blue, green
  - **CMYK**: cyan, magenta, yellow, black
  - **HSV**: hue, saturation, value
- Attributes have *invariants*
  - Limits the attribute values
  - Example: red is int in 0..255
  - Get an error if you violate



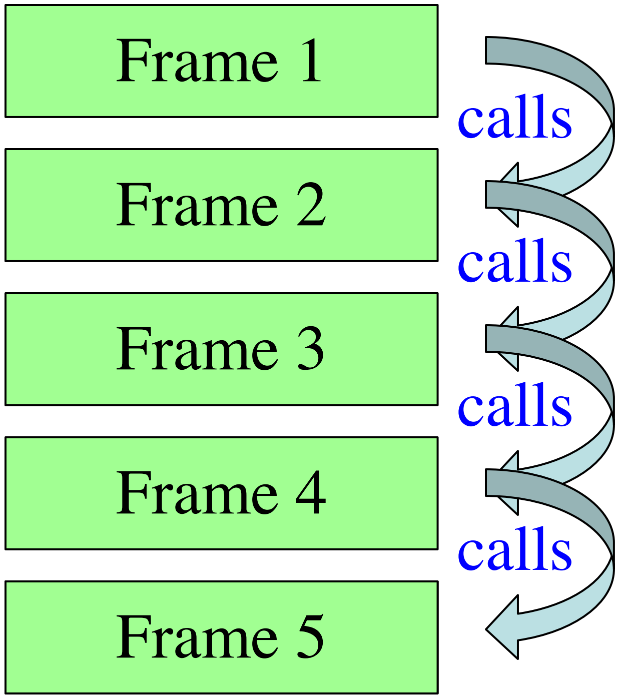
Constructor function. To make a **new** color.

```
>>> import introcs
>>> c = introcs.RGB(128,0,0)
>>> r = c.red
>>> c.red = 500 # o
AssertionError: 500
```

Accessing Attribute

Recall: The Call Stack

- Functions are **stacked**
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a ***high water mark***
  - Must have enough to keep the **entire stack in memory**
  - Error if cannot hold stack



Error Messages

Not An Error Message

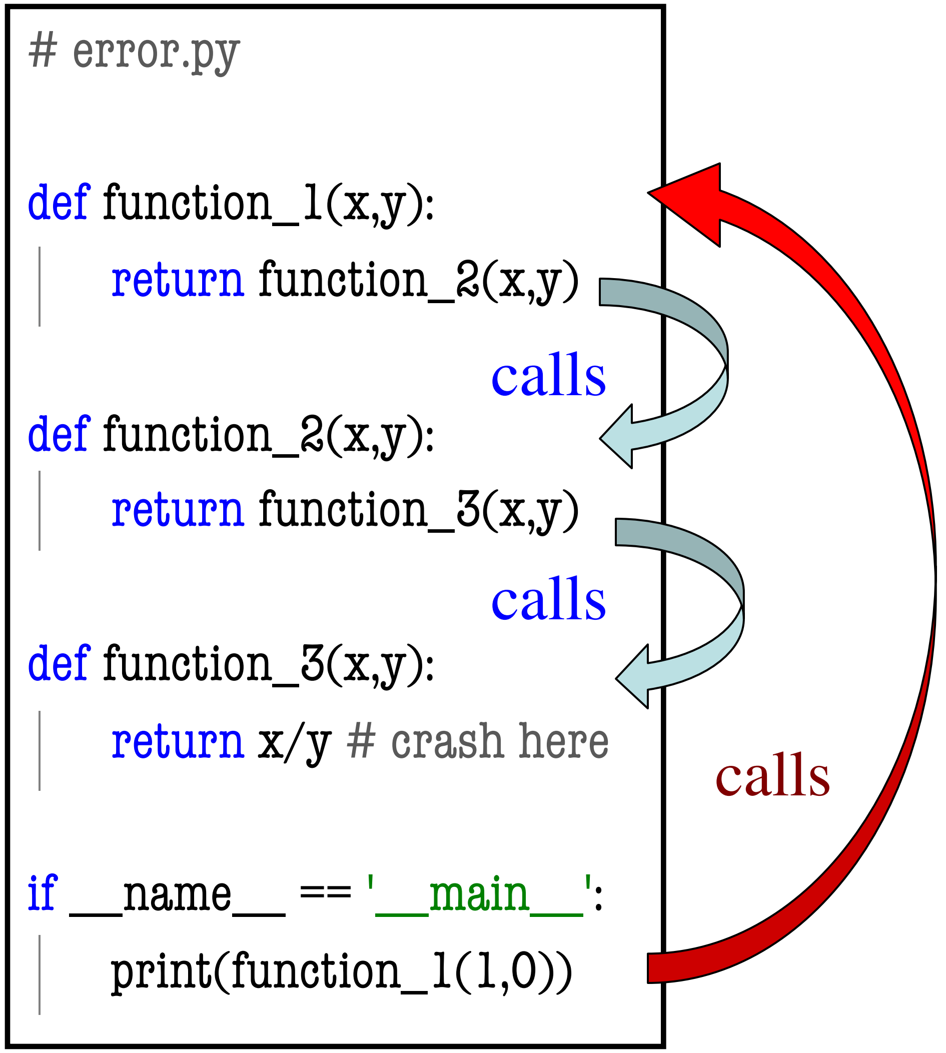
```
ZeroDivisionError: division by zero
```

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
```

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))
```

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
```

Make sure you can see line numbers in Atom.

Errors and the Call Stack

```
#
de
    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

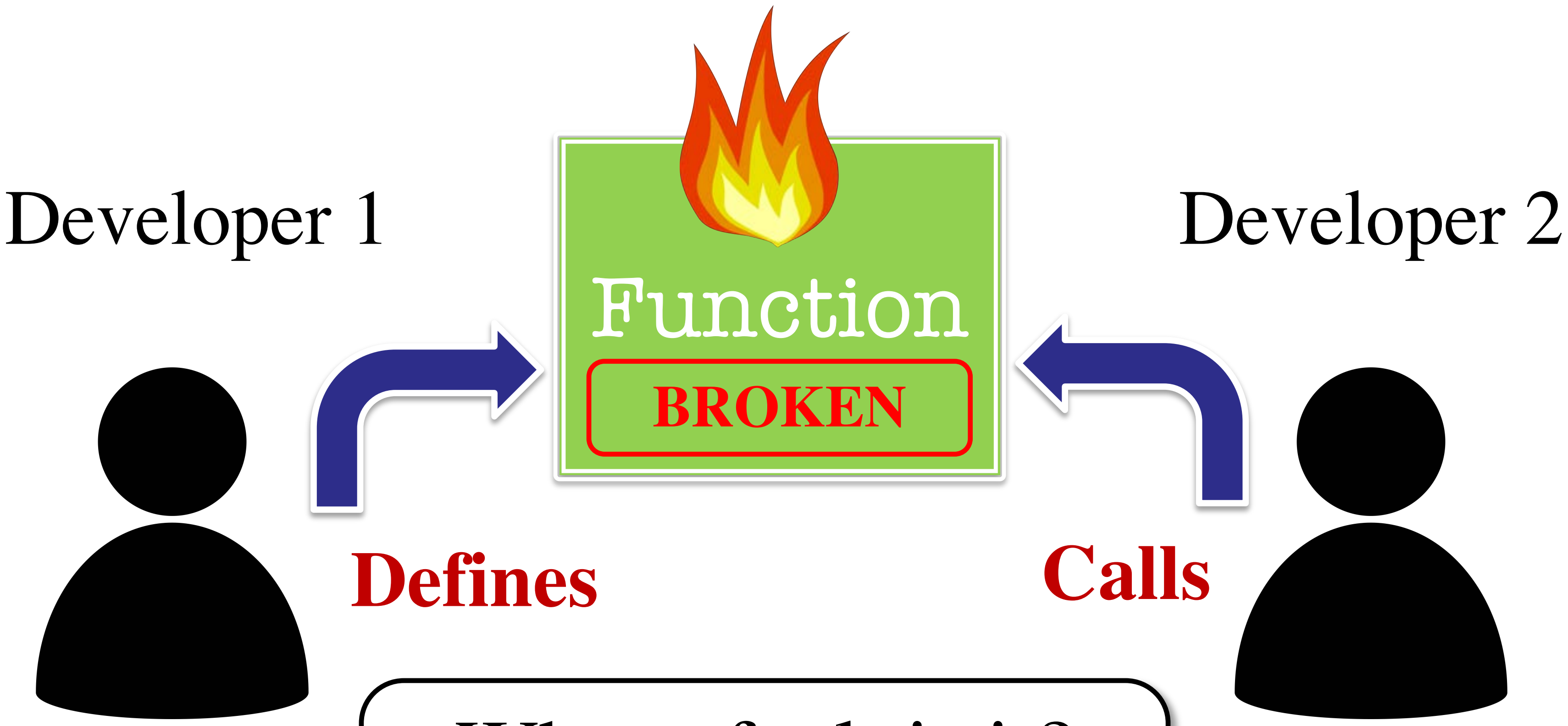
Crashes produce the call stack:

```
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
```

Where error occurred
(or where was found)

Make sure you can see line numbers in Atom.

Recall: Assigning Responsibility



Whose fault is it?
Who must fix it?

Determining Responsibility

```
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 Responsibility

```
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 A dule>
    print(function_1(1,0))

  File "error1.py", line 18, in B tion_1
    return function_2(x,y)

  File "error1.py", line 28, in C tion_2
    return x/y

ZeroDivision
```

Where is the error?

Determining Responsibility

```
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 Responsibility

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

  File "error1.py", line 18, in        tion_1
    return function_2(x,y)

  File "error1.py", line 28, in        tion_2
    return x/y

ZeroDivision
```

A

B

C

Where is the error?

Determining Responsibility

```
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 Responsibility

```
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

- Responsibility is "outside of Python"
  - 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 string
- Statement to **verify a fact is true**
  - Similar to `assert_equals` used in unit tests
  - But more versatile with complete **stack trace**

Why Do This?

- Enforce preconditions!
  - Put precondition as assert.
  - If violate precondition, the program crashes
- Provided code in A3 uses asserts heavily
  - First slide of lecture!

```
def exchange(from_c, to_c, amt)
    """Returns: amt from exchange
       Precondition: amt a float..."""
    assert type(amt) == float
    ...
```

Will do yourself in **A4**.

```
assert <boolean>              # Creates error if <boolean> false
assert <boolean>, <string>    # As above, but displays <String>
```

Example: Anglicizing an Integer

```
def anglicize(n):
    """Returns: the anglicization of int n.
    Precondition: n an int, 0 < n < 1,000,000"""
    assert type(n) == int, repr(n)+' is not an int'
    assert 0 < n and n < 1000000, repr(n)+' is out of range'
    # Implement method here...
```

Example: Anglicizing an Integer

```
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: n an int, 0 < n < 1,000,000"""
    assert type(n) == int, repr(n)+' is not an int'
    assert 0 < n and n < 1000000, repr(n)+' is out of range'
    # Implement method here...
```

Check (part of) the precondition

Error message when violated

Aside: Using **repr** Instead of **str**

```
>>> 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

Aside: Using **repr** Instead of **str**

```
>>> 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

Aside: Using **repr** Instead of **str**

```
>>> 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 lookup_netid(nid):
    """Returns: name of student with netid nid.

    Precondition: nid is a string, which consists of
    2 or 3 letters and a number"""
    assert ?????
```

Assert use expressions only.
Cannot use if-statements.
Each one must fit on one line.

Sometimes we will only enforce part of the precondition

Enforcing Preconditions is Tricky!

```
def lookup_netid(nid):
    """Returns: name of student with netid nid.

    Precondition: nid is a string, which consists of
    2 or 3 letters and a number"""
    assert type(nid) == str, repr(nid) + ' is not a string'
    assert nid.isalnum(), nid+' is not just letters/digits'
```

Returns True if s contains only letters, numbers.

Does this catch all violations?

Using Functions to Enforce Preconditions

```
def exchange(curr_from, curr_to, amt_from):
    """Returns: amount of curr_to received.

    Precondition: curr_from is a valid currency code
    Precondition: curr_to is a valid currency code
    Precondition: amt_from is a float"""
   assert ??????, repr(curr_from) + ' not valid'
   assert ??????, repr(curr_from) + ' not valid'
   assert type(amt_from)==float, repr(amt_from)+' not a float'
```

Using Functions to Enforce Preconditions

```
def exchange(curr_from, curr_to, amt_from):
    """Returns: amount of curr_to received.

    Precondition: curr_from is a valid currency code
    Precondition: curr_to is a valid currency code
    Precondition: amt_from is a float"""
    assert iscurrency(curr_from), repr(curr_from) + ' not valid'
    assert iscurrency(curr_to), repr(curr_to) + ' not valid'
    assert type(amt_from)==float, repr(amt_from)+' not a float'
```

Recovering from Errors

- Suppose we have this code:

```
result = input('Number: ')        # get number from user
x = float(result)                  # convert string to float
print('The next number is '+str(x+1))
```

- What if user mistypes?

```
Number: 12a
Traceback (most recent call last):
  File "prompt.py", line 13, in <module>
    x = float(result)
ValueError: could not convert string to float: '12a'
```

Ideally Would Handle with Conditional

```
result = input('Number: ')      # get number from user
if isfloat(result):

    x = float(result)            # convert to float
    print('The next number is '+str(x+1))
else:
    print('That is not a number!')
```

Does not Exist

Using Try-Except

```
try:
    result = input('Number: ')     # get number
    x = float(result)              # convert to float
    print('The next number is '+str(x+1))
except:
    print('That is not a number!')
```

Similar to if-else

- But always does the try block
- Might not do **all** of the try block

Using Try-Except

```
try:
    result = input('Number: ')     # ge
    x = float(result)              # convert to float
    print('The next number is '+str(x+1))
except:
    print('That is not a number!')
```

Conversion may crash!

Execute if crashes

Similar to if-else

- But always does the try block
- Might not do **all** of the try block

Try-Except is Very Versatile

```
def isfloat(s):
    """Returns: True if string
    s represents a float"""
    try:
        x = float(s)
        return True
    except:
        return False
```

Conversion to a float might fail

If attempt succeeds, string s is a float

Otherwise, it is not

Try-Except and the Call Stack

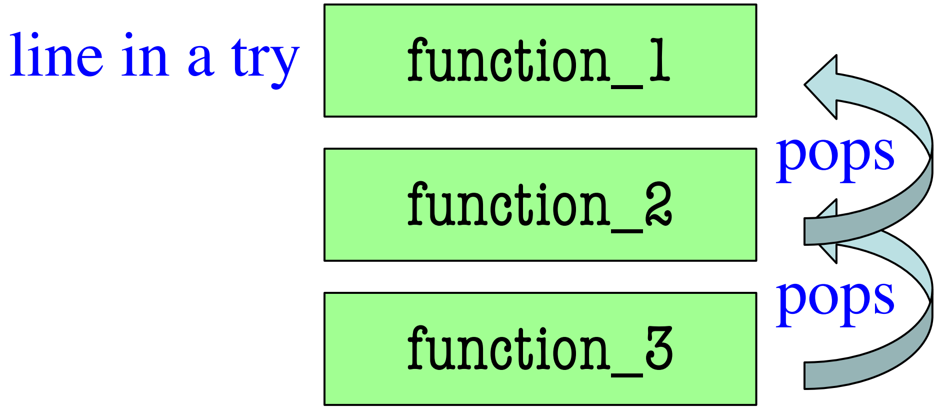
```
# recover.py

def function_1(x,y):
    try:
        return function_2(x,y)
    except:
        return float('inf')

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

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

- Error "pops" frames off stack
  - Starts from the stack bottom
  - Continues until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error



Try-Except and the Call Stack

```
# recover.py

def function_1(x,y):
    try:
        return function_2(
    except:
        return float('inf')

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

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

How to return
∞ as a float.

- Error "pops" frames off stack
  - om the stack bottom
  - es until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error
- **Example**:

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

No traceback!

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

```
'Starting first.'
'Starting second.'
'Starting third.'
'Caught at second'
'Ending second'
'Ending first'
```

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(0)?

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(0)?

```
'Starting first.'
'Starting second.'
'Starting third.'
'Ending third'
'Ending second'
'Ending first'
```