Lecture 11

Asserts and Error Handling
Announcements for Today

Reading

• Reread Chapter 3
• 10.0-10.2, 10.4-10.6 for Thu

Assignments

• Assignment 1 now complete
  ▪ Unless we gave extension
• Assignment 2 also graded
  ▪ Ready for pick-up Thurs
  ▪ Solutions posted in CMS
• Assignment 3 due next week
  ▪ Before you leave for break
  ▪ Same “length” as A1
  ▪ Get help now if you need it

• Prelim, Oct 15th 7:30-9:30
  ▪ Material up October 6th
  ▪ Study guide next week
• Conflict with Prelim time?
  ▪ Submit to Prelim 1 Conflict assignment on CMS
  ▪ Do not submit if no conflict
Using Color Objects in A3

- New classes in `colormodel`
  - 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

```python
>>> import colormodel
>>> c = colormodel.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 colormodel
  - 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

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

    """Returns: color rgb in space CMYK
    Precondition: rgb is an RGB object"
"
    # DO NOT CONSTRUCT AN RGB OBJECT
    # Variable rgb already has RGB object
    # 1. Access attributes from rgb folder
    # 2. Plug into formula provided
    # 3. Compute the new cyan, magenta, etc. values
    # 4. Construct a new CMYK object
    # 5. Return the newly constructed object

    Only time you will ever call a constructor
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.py

```python
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)
```

9/29/15

Asserts & Error Handling
Errors and the Call Stack

# error.py

```python
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 Komodo. Preferences ➔ Editor
Errors and the Call Stack

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 Komodo.
Preferences ➔ Editor
```


Assert Statements

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

• Way to force an error
  ▪ Why would you do this?
• Enforce preconditions!
  ▪ Put precondition as assert.
  ▪ If violate precondition, the program crashes
• Provided code in A3 uses asserts heavily

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

Will do yourself in A4.
Example: Anglicizing an Integer

```python
def anglicize(n):
    
    """Returns: the anglicization of int n.
    Precondition: n an int, 0 < n < 1,000,000"""

    assert type(n) == int, str(n)+' is not an int'
    assert 0 < n and n < 1000000, str(n)+' is out of range'

    # Implement method here...
```

9/29/15

Asserts & Error Handling
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, str(n)+' is not an int'
    assert 0 < n and n < 1000000, str(n)+' is out of range'
    # Implement method here...

Check (part of) the precondition

Error message when violated
Enforcing Preconditions is Tricky!

```python
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.
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, str(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 Function to Enforce Preconditions

```python
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 ????, str(curr_from) + ' not valid'
    assert ????, str(curr_from) + ' not valid'
    assert type(amt_from)==float, str(amt_from) + ' not a float'
```

9/29/15

Asserts & Error Handling
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), str(curr_from) + ' not valid'
    assert iscurrency(curr_to), str(curr_from) + ' not valid'
    assert type(amt_from)==float, str(amt_from) + ' not a float'
Recovering from Errors

- try-except blocks allow us to recover from errors
  - Do the code that is in the try-block
  - Once an error occurs, jump to the catch

- **Example**:

  ```python
  try:
      input = raw_input()  # get number from user
      x = float(input)     # convert string to float
      print 'The next number is ' + str(x+1)
  except:
      print 'Hey! That is not a number!'  
  ```

  - might have an error
  - executes if error happens
Recovering from Errors

- try-except blocks allow us to recover from errors:
  - Do the code that is in the try block
  - Once an error occurs, jump to the except block
- Example:

```python
try:
    input = raw_input()  # get number from user
    x = float(input)     # convert string to float
    print 'The next number is ' + str(x+1)
except:
    print 'Hey! That is not a number!'
```

Similar to if-else:
- But always does try
- Just might not do all of the try block
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

• 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

```python
# 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
```
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

How to return $\infty$ as a float.

- Error “pops” frames off stack from the stack bottom to the current line is in a try-block
- Jumps to except, and then proceeds as if no error

**Example:**

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

No traceback!
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)?
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'
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)?
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'