Lecture 10

The Call Stack
Announcements for Today

Reading

- 10.0-10.2, 10.4-10.6 for Tue

Assignments

- Work on your revisions
  - Want done by Sunday
- Survey: 418 responded
  - If not responded, do Sunday
  - Avg Time: 6.5 hours
  - STD Dev: 4 hours
- Assignment 2 also Sunday
  - Scan and submit online
- Assignment 3 up Monday
Modeling Storage in Python

• **Global Space**
  - What you “start with”
  - Stores global variables
  - Also *modules & functions!*
  - Lasts until you quit Python

• **Call Frame**
  - Variables in function call
  - Deleted when call done

• **Heap Space**
  - Where “folders” are stored
  - Have to access indirectly
Frames and Helper Functions

```python
def last_name_first(s):
    """Precondition: s in the form <first-name> <last-name>""
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first

def first_name(s):
    """Prec: see last_name_first""
    end = s.find(' ')
    return s[0:end]
```

Call: last_name_first('Walker White'):
def last_name_first(s):
    """Precondition: s in the form <first-name> <last-name>""
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first

def first_name(s):
    """Prec: see last_name_first""
    end = s.find(' ')
    return s[0:end]
def last_name_first(s):
    
    """**Precondition**: s in the form <first-name> <last-name>"""

    first = first_name(s)

    last = last_name(s)

    return last + ',' + first


def first_name(s):
    
    """**Prec**: see last_name_first"""

    end = s.find(' ')

    return s[0:end]
Frames and Helper Functions

```python
def last_name_first(s):
    """Precondition: s in the form <first-name> <last-name>""
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first

def first_name(s):
    """Prec: see last_name_first""
    end = s.find(' ')
    return s[0:end]
```

Call: `last_name_first('Walker White')`:

```
last_name_first
1

s
'Walker White'
```

```
first_name

s
'Walker White'
end
6
RETURN
'Walker'
```
Frames and Helper Functions

```python
def last_name_first(s):
    '''Precondition: s in the form
    <first-name> <last-name>'''
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first

def first_name(s):
    '''Prec: see last_name_first'''
    end = s.find(' ')
    return s[0:end]
```

Call: last_name_first('Walker White'):
Frames and Helper Functions

```python
def last_name_first(s):
    """Precondition: s in the form <first-name> <last-name>""
    first = first_name(s)
    last = last_name(s)
    return last + '.' + first

def last_name(s):
    """Prec: see last_name_first""
    end = s.find(' ')
    return s[end+1:]
```

Call: last_name_first('Walker White'):

```
Call Stack
<table>
<thead>
<tr>
<th>Method</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_name_first</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>first</td>
<td>1</td>
</tr>
<tr>
<td>Walker</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>Walker White</td>
<td></td>
</tr>
</tbody>
</table>
```

9/25/14
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
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
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
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
The Call Stack

- Functions are “stacked”
  - Cannot remove one above without 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

Book adds a special “frame” called module. This is **WRONG**! Module is global space.
Errors and the Call Stack

```python
# 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)
```

The Call Stack

9/25/14
Errors and the Call Stack

Crashes produce the call stack:

```python
# 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)
```

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

Script code. Global space

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

Where error occurred (or where was found)

Make sure you can see line numbers in Komodo.
Preferences ➔ Editor
**Assert Statements**

- **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 this in A3.
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...
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...
```

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 only enforce part of the precondition
```

9/25/14 The Call Stack 21
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?
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 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!' 
  ```

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

```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
```

- 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(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 from the stack bottom
  - 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

- **Example:**
  ```
  >>> 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)?
```python
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)?
Tracing Control Flow

```python
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'