Lecture 15

More Recursion
Announcements for This Lecture

Prelim 1

- Prelim 1 available
  - Pick up in Lab Section
  - Solution posted in CMS
  - Mean: 84, Median: 87
- What are letter grades?
  - Way too early to tell
  - A: Could be a consultant
  - B: Could take 2110
  - C: Good enough to pass

Assignments and Labs

- Need to be working on A4
  - Instructions are posted
  - Just reading it takes a while
  - Slightly longer than A3
  - Problems are harder
- Lab Today: lots of practice!
  - 5 functions are mandatory
  - Lots of optional ones to do
  - Exam questions on Prelim 2
Recursion

- **Recursive Definition:**
  A definition that is defined in terms of itself
- **Recursive Function:**
  A function that calls itself (directly or indirectly)

- **Powerful programming tool**
  - Want to solve a difficult problem
  - Solve a simpler problem instead

- **Goal of Recursion:**
  Solve original problem with help of simpler solution
Example: Reversing a String

• Precise Specification:
  ▪ Returns: reverse of s

• Solving with recursion
  ▪ Suppose we can reverse a smaller string (e.g. less one character)
  ▪ Can we use that solution to reverse whole string?

• Often easy to understand first without Python
  ▪ Then sit down and code
Example: Reversing a String

- Precise Specification:
  - Returns: reverse of s

- Solving with recursion
  - Suppose we can reverse a smaller string (e.g., less one character)
  - Can we use that solution to reverse whole string?

- Often easy to understand first without Python
  - Then sit down and code
Example: Reversing a String

• Precise Specification:
  ▪ Returns: reverse of s

• Solving with recursion
  ▪ Suppose we can reverse a smaller string (e.g. less one character)
  ▪ Can we use that solution to reverse whole string?

• Often easy to understand first without Python
  ▪ Then sit down and code
Example: Reversing a String

• Precise Specification:
  ▪ Returns: reverse of s

• Solving with recursion
  ▪ Suppose we can reverse a smaller string (e.g. less one character)
  ▪ Can we use that solution to reverse whole string?

• Often easy to understand first without Python
  ▪ Then sit down and code
def reverse(s):
    """Returns: reverse of s
    ""
    # {s is empty}
    if s == '':
        return s
    # {s at least one char}
    # (reverse of s[1:]) + s[0]
    return reverse(s[1:]) + s[0]
Example: Palindromes

• String with $\geq 2$ characters is a palindrome if:
  ▪ its first and last characters are equal, and
  ▪ the rest of the characters form a palindrome

• Example:

  have to be the same

  AMANAPLANACANALPANAMA

  has to be a palindrome

• Precise Specification:

  ```python
def ispalindrome(s):
      """Returns: True if s is a palindrome"""
  ```
Example: Palindromes

- String with $\geq 2$ characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome

- Recursive Function:

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True
    else:
        return s[0] == s[-1] and ispalindrome(s[1:-1])
```

Recursive Definition

Recursive case

Base case
Example: Palindromes

- String with ≥ 2 characters is a palindrome if:
  - its first and last characters are equal,
  - the rest of the characters form a palindrome.

- Recursive Function:
  ```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
      return True
    return s[0] == s[-1] and ispalindrome(s[1:-1])
  ```

1. Precise specification?
2. Base case: correct?
3. Recursive case: progress to termination?
4. Recursive case: correct?
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    // { s has at least two characters }
    return (equals_ignore_case(s[0],s[-1])
        and ispalindrome2(s[1:-1]) )
```python
def ispalindrome2(s):
    
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True

    // { s has at least two characters }
    return (equals_ignore_case(s[0],s[-1])
             and ispalindrome2(s[1:-1]) )
```

Example: More Palindromes

Precise Specification
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    return (equals_ignore_case(s[0], s[-1])
            and ispalindrome2(s[1:-1]) )

def equals_ignore_case(a, b):
    """Returns: True if a and b are same ignoring case"""
    return a.upper() == b.upper()
Example: More Palindromes

```python
def ispalindrome3(s):
    """Returns: True if s is a palindrome
    Case of characters and non-letters ignored."""
    return ispalindrome2(depunct(s))

def depunct(s):
    """Returns: s with non-letters removed""
    if s == ":
        return s
    # use string.letters to isolate letters
    return (s[0]+depunct(s[1:]) if s[0] in string.letters
            else depunct(s[1:]))
```

Use helper functions!
- Often easy to break a problem into two
- Can use recursion more than once to solve
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"
```

**Approach 1**

```
5
341267
```
def commafy(s):
    '''Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int''''

Approach 1

5 341267

commafy

341,267
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

**Approach 1**

```
5  341267
   ↓ commafy
5  341,267
```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""

**Approach 1**

5 341267

| Always? When? | 5 , 341,267 | commafy | 5 341267 |
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""

Approach 1

5
341267

Approach 2

5341
267

Always? When?
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""
```

**Approach 1**

5

341267

```
5
, 341,267
```

**Approach 2**

5341

267

```
5,341
```

Always? When?
def commafy(s):
    ""
    Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"

Approach 1

Approach 2

Always? When?
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""
```

**Approach 1**

1. `5`
2. `341267`
3. `commafy`
4. `5`, `341,267`

**Approach 2**

1. `5341`
2. `267`
3. `commafy`
4. `5,341`, `267`

Always? When?

10/2/12

More Recursion

Always!
def commafy(s):
    """Returns: string with commas every 3 digits
e.g. commafy('5341267') = '5,341,267'
Precondition: s represents a non-negative int"""
    # No commas if too few digits.
    if len(s) <= 3:
        return s
    # Add the comma before last 3 digits
    return commafy(s[:-3]) + ',' + s[-3:]
How to Break Up a Recursive Function?

```python
def exp(b, c):
    """Returns: b^c
    Precondition: b a float, c ≥ 0 an int"""
```

**Approach 1**

\[
12^{256} = 12 \times (12^{255})
\]

\[
b^c = b \times (b^{c-1})
\]

**Approach 2**

\[
12^{256} = (12^{128}) \times (12^{128})
\]

\[
b^c = (b \times b)^{c/2} \text{ if } c \text{ even}
\]
Raising a Number to an Exponent

**Approach 1**

```python
def exp(b, c):
    """Returns: \(b^c\)
    Precondition: \(b\) a float,
    \(c \geq 0\) an int"
    # \(b^0\) is 1
    if c == 0:
        return 1
    # \(b^c = b(b^c)\)
    return b*exp(b,c-1)
```

**Approach 2**

```python
def exp(b, c):
    """Returns: \(b^c\)
    Precondition: \(b\) a float,
    \(c \geq 0\) an int"
    if c == 0:
        return 1
    # \(c > 0\)
    if c % 2 == 0:
        return exp(b*b,c/2)
    return b*exp(b*b,c/2)
```
def exp(b, c):
    """Returns: \( b^c \)"
    Precondition: \( b \) a float,
    \( c \geq 0 \) an int"
    # \( b^0 \) is 1
    if c == 0:
        return 1
    # \( c > 0 \)
    if c % 2 == 0:
        return exp(b*b, c/2)
    return b*exp(b*b, c/2)

<table>
<thead>
<tr>
<th>c</th>
<th># of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>n + 1</td>
</tr>
</tbody>
</table>

32768 is 215
\( b^{32768} \) needs only 215 calls!
Recursion and Objects

- Class Person (person.py)
  - Objects have 3 attributes
    - name: String
    - mom: Person (or None)
    - dad: Person (or None)
  - Represents the “family tree”
    - Goes as far back as known
    - Attributes mom and dad are None if not known
- Constructor: Person(n,m,d)
  - Or Person(n) if no mom, dad
def num_ancestors(p):
    '''Returns: num of known ancestors
    Pre: p is a Person'''
    # Base case
    # No mom or dad (no ancestors)
    # Recursive step
    # Has mom or dad
    # Count ancestors of each one
    # (plus mom, dad themselves)
    # Add them together
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""

    # Base case
    if p.mom == None and p.dad == None:
        return 0

    # Recursive step
    moms = 0
    if not p.mom == None:
        moms = 1 + num_ancestors(p.mom)
    dads = 0
    if not p.dad == None:
        dads = 1 + num_ancestors(p.dad)
    return moms + dads

11 ancestors
Space Filling Curves

Challenge

• Draw a curve that
  ▪ Starts in the left corner
  ▪ Ends in the right corner
  ▪ Touches every grid point
  ▪ Does not touch or cross itself anywhere

• Useful for analysis of 2-dimensional data

Starts Here

Ends Here
Hilbert’s Space Filling Curve

Hilbert(1):

Hilbert(2):

Hilbert(n):

H(n-1) down

H(n-1) down

H(n-1) left

H(n-1) right

More Recursion
Hilbert’s Space Filling Curve

Basic Idea

• Given a box
• Draw $2^n \times 2^n$ grid in box
• Trace the curve
• As $n$ goes to $\infty$, curve fills box
“Turtle” Graphics: Assignment A4