Recursion and Iteration

- Recursion *theoretically equivalent* to iteration
  * Anything can do in one, can do in other
  * But what is easy in one may be hard in other
  * **When is using recursion better?**
- Recursion is more *flexible in breaking up data*
  * Iteration typically scans data left-to-right
  * Recursion works with other “slicings”
- Recursion has *interesting advanced applications*
  * See some of these in Assignment 4

Example: Palindromes

- String with ≥ 2 characters is a palindrome if:
  * its first and last characters are equal, and
  * the rest of the characters form a palindrome
- **Example:**
  
  \[\text{AMANAPLANACANALPANAMA}\]
  
  has to be a palindrome
- **Precise Specification:**
  
  ```python
  def is_palindrome(s):
      """Returns: True if s is a palindrome""
      if len(s) < 2:
          return True
      # { s has at least two characters }
      return s[0] == s[-1] and is_palindrome(s[1:-1])
  ```

Example: More Palindromes

```python
def is_palindrome2(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 is_palindrome2(s[1:-1])

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

Recursion is form of Divide and Conquer

**Goal:** Solve problem P on a piece of data

**Idea:** Split data into two parts and solve problem

<table>
<thead>
<tr>
<th>data</th>
<th>data 1</th>
<th>data 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve Problem P</td>
<td>Solve Problem P</td>
<td>Combine Answer!</td>
</tr>
</tbody>
</table>

Where work is all done

How to Break Up a Recursive Function?

**Approach 1**

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy(5341267) = 5,341,267"
    if s == '':
        return ''
    # Precondition: s represents a non-negative int
    return commafy(s[0:3]) + (',' + commafy(s[3:]))
```

**Approach 2**

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy(5341267) = 5,341,267"
    if s == '':
        return ''
    # Precondition: s represents a non-negative int
    return commafy(s[0:3]) + (',' + commafy(s[3:]))
```

<table>
<thead>
<tr>
<th>Approach 1</th>
<th>Approach 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 341 267</td>
<td>5341 267</td>
</tr>
<tr>
<td>commafy</td>
<td>commafy</td>
</tr>
<tr>
<td>Always? Yes</td>
<td>Always? Yes</td>
</tr>
</tbody>
</table>
How to Break Up a Recursive Function?

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:]

Raising a Number to an Exponent

def exp(b, c):
    """Returns: b^c
    Precondition: b a float, c >= 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)

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

How to Break Up a Recursive Function?

def exp(b, c):
    """Returns: b^c
    Precondition: b a float, c >= 0 an int"""

    Approach 1
    12^256 = 12 \times (12^{255})
    Recursive

    Approach 2
    12^256 = (12^128) \times (12^{128})
    Recursive
    Recursive
    Recursive

    b^c = b \times (b^{c-1})
    b^c = (b \times b)^{c/2} if c even

Hilbert’s Space Filling Curve

2^n

Hilbert(1):

Hilbert(2):

Hilbert(n):
• Hilbert(n-1) down
• Hilbert(n-1) right

Recursion and Objects

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
    mem = 0
    if not p.mom == None:
        mem = 1 + num_ancestors(p.mom)
    dads = 0
    if not p.dad == None:
        dads = 1 + num_ancestors(p.dad)
    return mem + dads

32768 is 215 calls!

11 ancestors