def reverse(s):
    """Returns: reverse of s
Precondition: s a string"""
    # 1. Handle small data
    if len(s) <= 1:
        return s

    # 2. Break into two parts
    left = s[0]
    right = reverse(s[1:])

    # 3. Combine the result
    return right + left
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?

Approach 2

5341
267

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""
    # 1. Handle small data.
    if len(s) <= 3:
        return s
    # 2. Break into two parts
    left = commafy(s[:-3])
    right = s[-3:]  # Small part on RIGHT
    # 3. Combine the result
    return left + ',' + right
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**

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

**Approach 2**

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

- Recursive
- Recursive
- Recursive

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

\[ b^c = (b\times b)^{c/2} \text{ if } c \text{ even} \]
def exp(b, c):
    """Returns: \( b^c \)
    Precond: 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-1)/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
Recursion and Objects

def num_ancestors(p):
    '''Returns: num of known ancestors
    Pre: p is a Person'''
    # 1. Handle small data.
    if p.mom == None and p.dad == None:
        return 0

    # 2. Break into two parts
    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)

    # 3. Combine the result
    return moms+dads

11 ancestors
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:

  AMANAPLANACANALPANAMA

  has to be a palindrome

  have to be the same

• Function to Implement:

  ```python
  def is_palindrome(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

def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True
    # Halves not the same; not divide and conquer
    ends = s[0] == s[-1]
    middle = ispalindrome(s[1:-1])
    return ends and middle
Recursive Functions and Helpers

def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    # Halves not the same; not divide and conquer
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrom(e(s[1:-1]))
    return ends and middle

def equals_ignore_case(a, b):
    """Returns: True if a and b are same ignoring case"""
    return a.upper() == b.upper()
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
    # Combine left and right
    if s[0] in string.letters:
        return s[0]+depunct(s[1:])
    # Ignore left if it is not a letter
    return depunct(s[1:]),

Use helper functions!
- Sometimes the helper is a recursive function
- Allows you break up problem in smaller parts
Hilbert’s Space Filling Curve

Hilbert(1):

Hilbert(2):

Hilbert(n):

\[ 2^n \]

\[ 2^n \]