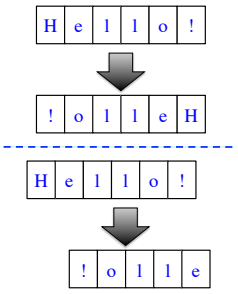
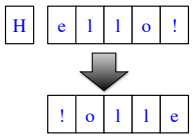
Example: Reversing a String

- **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

```
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # {s is empty}
    if s == '':
        return s

    # { s at least one char }
    # (reverse of s[1:])+s[0]
    return reverse(s[1:])+s[0]
```



✔ 1. Precise specification?
✔ 2. Base case: correct?
✔ 3. Recursive case: progress to termination?
✔ 4. Recursive case: correct?

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:**

have to be the same

AMANAPLANACANALPANAMA

has to be a palindrome

- **Precise Specification:**

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

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
- **Recursive Function:**

Recursive Definition

```
def ispalindrome(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 ispalindrome(s[1:-1])
```

Base case

Recursive case

Example: More Palindromes

```
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]) )

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

Precise Specification

Example: More Palindromes

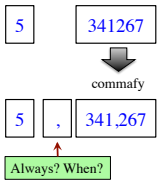
```
def ispalindrome3(s):
    """Returns: True if s is a palindrome
    Case of characters and non-letters ignored."""
    return ispalindrome2(depunct(s))

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

Use helper functions!
- Often easy to break a problem into two
- Can use recursion more than once to solve

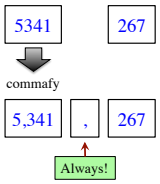
How to Break Up a Recursive Function?

```
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



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

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

Approach 2

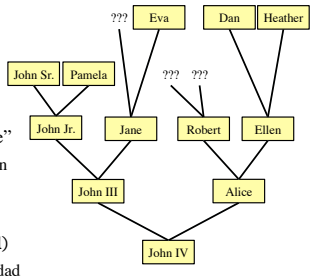
$12^{256} = (12^{128}) \times (12^{128})$

Recursive Recursive

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

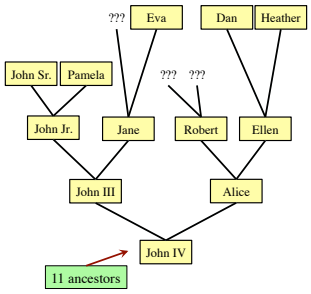
Recursion and Objects

- 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

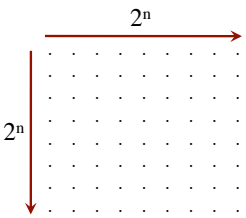


Recursion and Objects

```
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 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
```



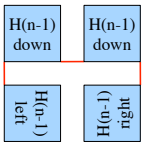
Hilbert's Space Filling Curve



Hilbert(1):

Hilbert(2):

Hilbert(n):



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 ∞, curve fills box