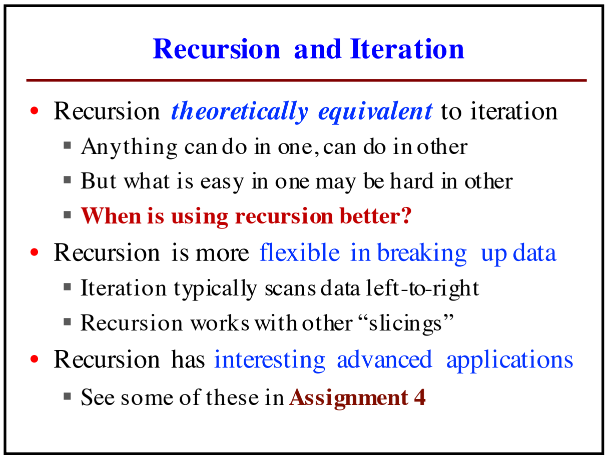
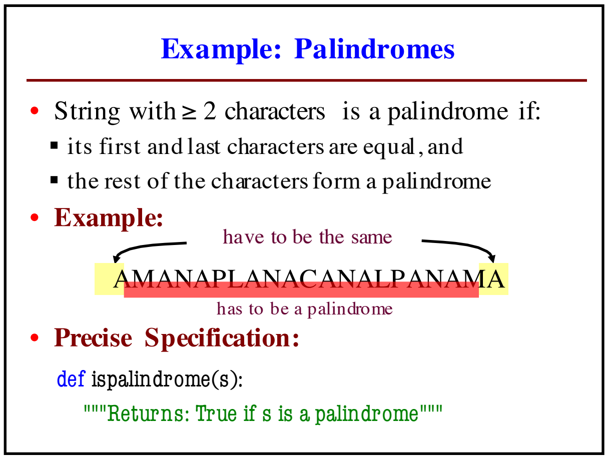


Recursion and Iteration

- 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**

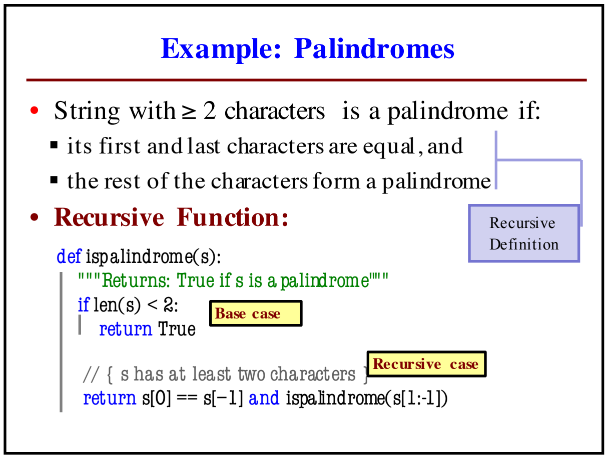


Example: Palindromes

- 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:**

- **Precise Specification:**

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

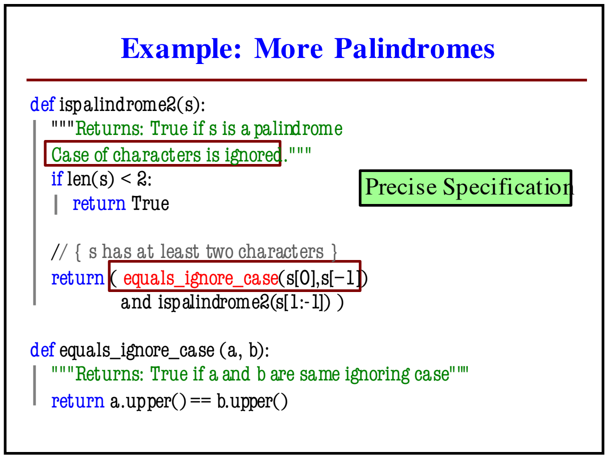


Example: Palindromes

- 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
Base case
Recursive case

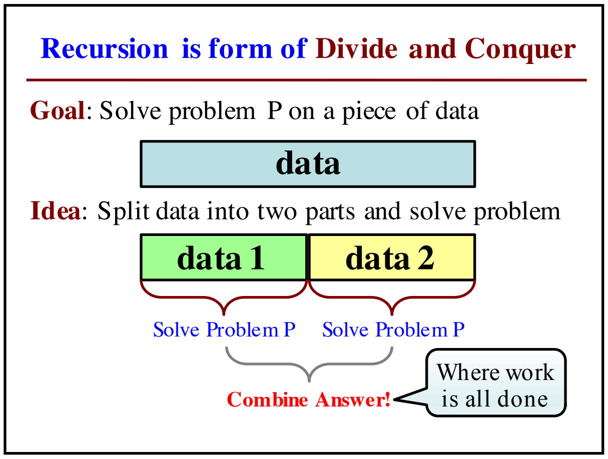
```
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True

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



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()
```

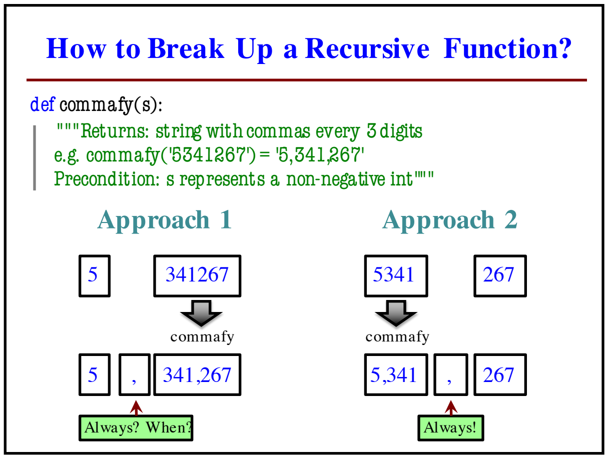
Precise Specification



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



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"""
```

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:]
```

Base case

Recursive case

How to Break Up a Recursive Function?

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

Approach 1

$$12^{256} = 12 \times (\underbrace{12^{255}}_{\text{Recursive}})$$

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

Approach 2

$$12^{256} = (\underbrace{12^{128}}_{\text{Recursive}}) \times (\underbrace{12^{128}}_{\text{Recursive}})$$

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

Raising a Number to an Exponent

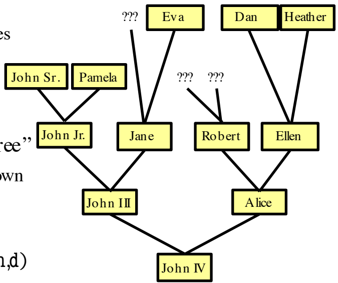
```
def exp(b, c)
    """Returns: bc
    Precondition: b a float,
                  c ≥ 0 an int"""
    # b0 is 1
    if c == 0:
        return 1

    # c > 0
    if c % 2 == 0:
        return exp(b*b,c/2)

    return b*exp(b*b,c/2)
```

c	# of calls
0	0
1	1
2	2
4	3
8	4
16	5
32	6
2^n	n + 1

32768 is 215
b^{32768} needs only 215 calls!

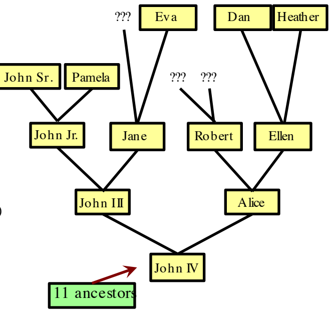
Recursion and Objects

- 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

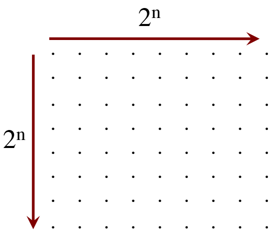


Recursion and Objects

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

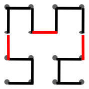


Hilbert's Space Filling Curve

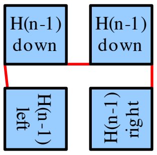


Hilbert(1):

Hilbert(2):



Hilbert(n):


H(n-1) down
H(n-1) down
H(n-1) left
H(n-1) right