RECURSION

(download slides and .py files to follow along)

6.100L Lecture 15

Ana Bell

ITERATIVE ALGORITHMS SO FAR

- Looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- Can capture computation in a set of **state variables** that update, based on a set of rules, on each iteration through loop
  - What is **changing each time** through loop, and how?
  - How do I **keep track** of number of times through loop?
  - When can I **stop**?
  - Where is the **result** when I stop?

MULTIPLICATION

- The * operator does this for us
- Make a function

```
def mult(a, b):
    return a*b
```

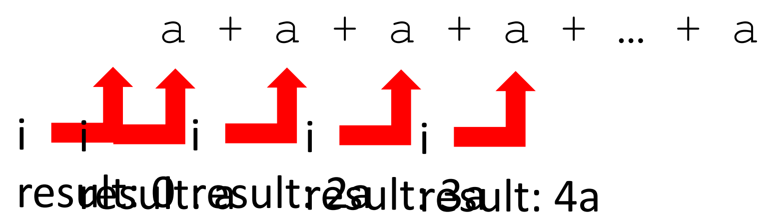
MULTIPLICATION
THINK in TERMS of ITERATION

- Can you make this iterative?
- Define `a*b` as `a+a+a+a... b` times
- Write a function

```
def mult(a, b):
    total = 0
    for n in range(b):
        total += a
    return total
```

MULTIPLICATION – ANOTHER ITERATIVE SOLUTION

- "multiply `a` * `b`" is equivalent to "add `b` copies of `a`"



- Capture **state** by

Update rules

  - An **iteration** number (`i`) starts at b
    `i` ← `i-1` and stop when 0
  - A current **value of computation** (`result`) starts at 0
    `result` ← `result + a`

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

MULTIPLICATION
NOTICE the RECURSIVE PATTERNS

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4 is 5+5***3**
  - But this is 5+5+5***2**
  - And this is 5+5+5+5***1**

Original problem

A very similar problem with one small change

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(     5*3     )`
  - `= 5+(5+(   5*2   ))`
  - `= 5+(5+(5+(5*1)))`

Original problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+( 5*3 )`
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+(5*1)))`

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(    5*3    )`
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+(5*1)))`

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION REACHED the END

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+( 5*3 )`
  - `= 5+(5+( 5*2 ))`
  - `= 5+(5+(5+(5*1)))`

Basic fact: a number multiplied with itself is the same number.

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - $5*4$
  - $= 5+(5*3)$
  - $= 5+(5+(\boxed{5*2}))$
  - $= 5+(5+(\boxed{5+(5)}))$

Similar problem

10

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+( 5*3 )`
  - `= 5+(5+( 10 ))`
  - `= 5+(5+(5+( 5 )))`

Similar problem

15

MULTIPLICATION
BUILD the RESULT BACK UP

- Recognize that we have a problem we are solving many times
- If **a = 5** and **b = 4**
  - 5*4 is   5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - `5*4`
  - `= 5+(      15       )`
  - `= 5+(5+(   10     ))`
  - `= 5+(5+(5+( 5 )))`

Original problem

20

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations

$$\boxed{a*b} = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations
- **Base case**
  - Keep reducing problem until reach a simple case that can be **solved directly**
  - When `b=1, a*b=a`

$$\boxed{\texttt{a*b}} = \underbrace{\texttt{a + a + a + a + ... + a}}_{\text{b times}}$$

$$= \texttt{a + }\underbrace{\texttt{a + a + a + ... + a}}_{\text{b-1 times}}$$

$$= \texttt{a + }\boxed{\texttt{a * (b-1)}}$$

recursive reduction

MULTIPLICATION – RECURSIVE CODE [Python Tutor LINK]

- **Recursive step**
  - `If b != 1, a*b = a + a*(b-1)`
- **Base case**
  - `If b = 1, a*b = a`

```python
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b-1)
```

base case

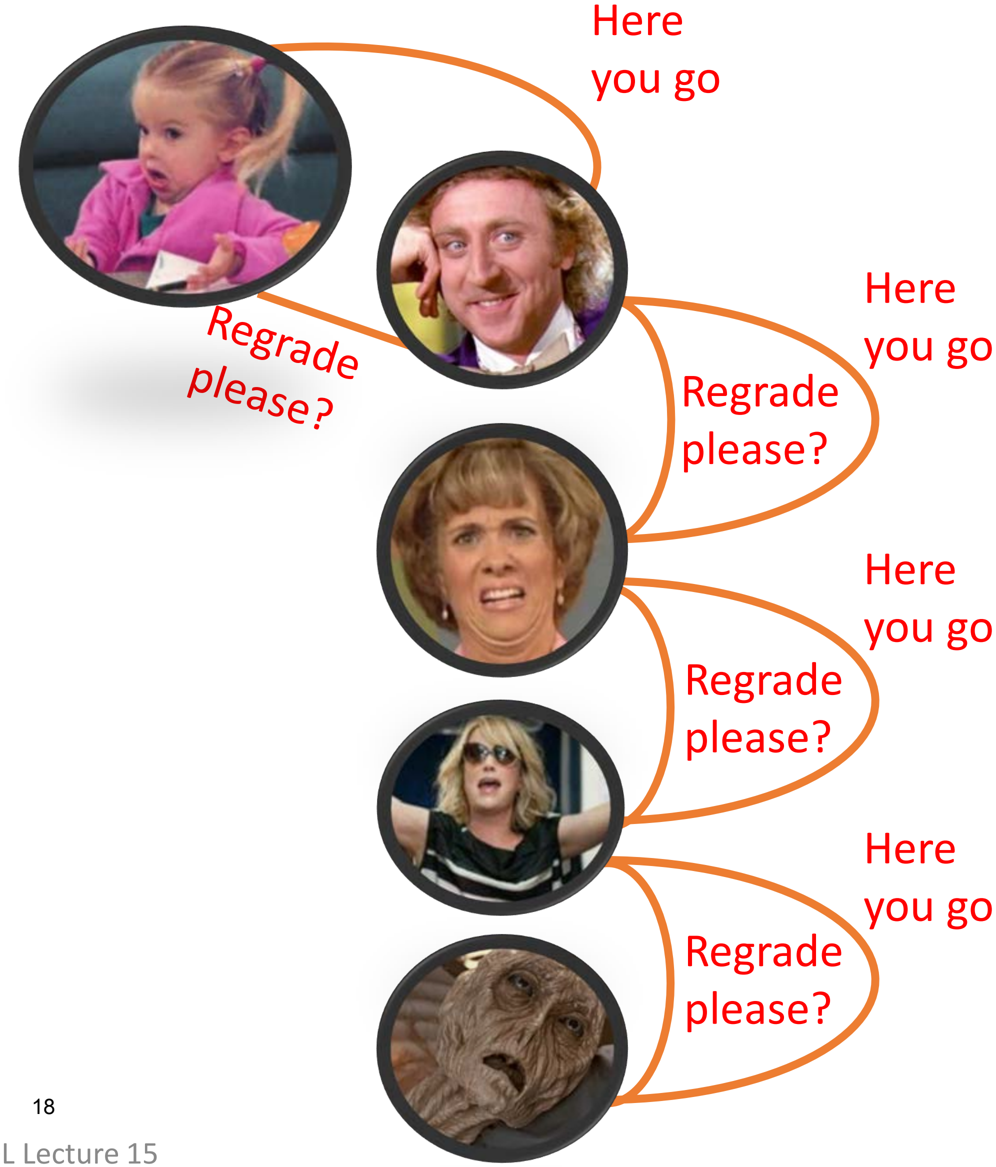
recursive step

REAL LIFE EXAMPLE
Student requests a regrade: **ONLY ONE** **function call**

Iteratives:

- Student asks the prof then the TA then the LA then the grader **one-by-one until one or more regrade the exam/parts**
- Student iterates through everyone and keeps track of the new score

REAL LIFE EXAMPLE
Student requests a regrade: **MANY function calls**



Recursive:

- 1) Student request(a **function call** to regrade!):
  - Asks the prof to regrade
  - Prof asks a TA to regrade
  - TA asks an LA to regrade
  - LA asks a grader to regrade

- 2) Relay the results (**functions return results** to their callers):
  - Grader tells the grade to the LA
  - LA tells the grade to the TA
  - TA tells the grade to the prof
  - Prof tells the grade to the student

BIG IDEA

"Earlier" function calls are waiting on results before completing.

WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - Reduce a problem to simpler versions of the same problem or to problem that can be solved directly
- Semantically: a programming technique where a **function calls itself**
  - In programming, goal is to NOT have infinite recursion
  - Must have **1 or more base cases** that are easy to solve directly
  - Must solve the same problem on **some other input** with the goal of simplifying the larger input problem, ending at base case

YOU TRY IT!

- Complete the function that calculates n^p for variables n and p

```
def power_recur(n, p):
    if _______:
        return ______
    elif _______:
        return ______
    else:
        return _________________
```

FACTORIAL

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

- For what `n` do we know the factorial?

  n = 1 →

  ```
  if n == 1:
      return 1
  ```

  base case

- How to reduce problem? Rewrite in terms of something simpler to reach base case

  n*(n-1)! →

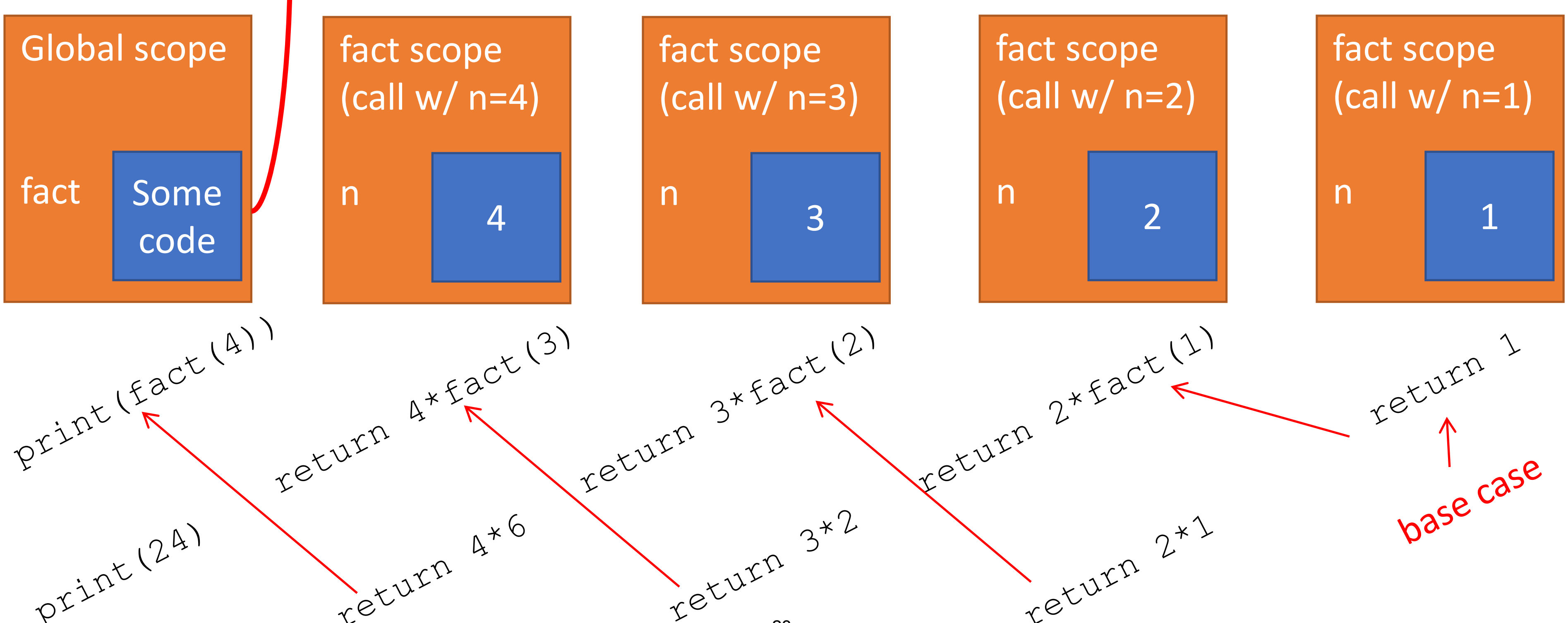
  ```
  else:
      return n*fact(n-1)
  ```

  recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
```



BIG IDEA

In recursion, each function call is completely separate.

Separate scope/environments.

Separate variable names.

Fully I-N-D-E-P-E-N-D-E-N-T

SOME OBSERVATIONS

Python Tutor LINK for factorial

- Each recursive call to a function creates its **own scope/environment**
- **Bindings of variables** in a scope are not changed by recursive call to same function
- Values of variable binding **shadow bindings** in other frames
- Flow of control passes back to **previous scope** once function call returns value

Using the same variable names but they are different objects in separate scopes

ITERATION vs. RECURSION

This version is much more Pythonic!

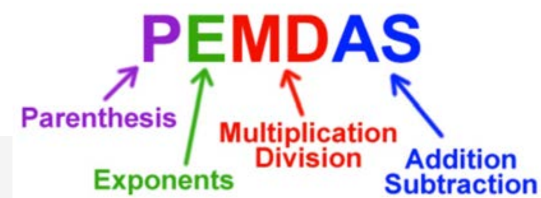
```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

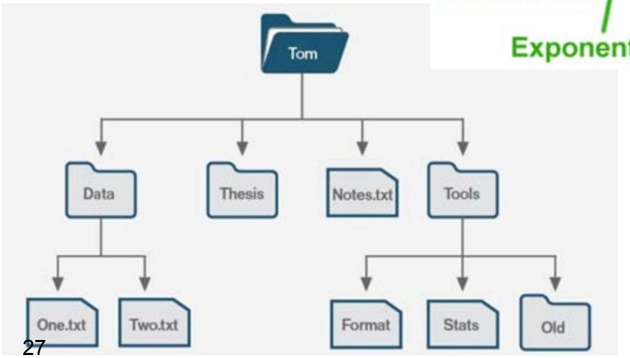
```
def fact_recur(n):
    if n == 1:
        return 1
    else:
        return n*fact_recur(n-1)
```

- Recursion may be efficient from programmer POV
- Recursion may not be efficient from computer POV

WHEN to USE RECURSION?
SO FAR WE SAW VERY SIMPLE CODE

- **Multiplication of two numbers** did not need a recursive function, did not even need an iterative function!
- Factorial was a little more intuitive to implement with recursion
  - We translated a mathematical equation that told us the structure
- **MOST problems do not need recursion** to solve them
  - If iteration is more intuitive for you then solve them using loops!
- **SOME problems yield far simpler code** using recursion
  - Searching a file system for a specific file
  - Evaluating mathematical expressions that use parens for order of ops





SUMMARY

- Recursion is a
  - Programming method
  - Way to divide and conquer
- A **function calls itself**
- A problem is broken down into a **base case** and a **recursive step**
- A base case
  - **Something you know**
  - You'll **eventually reach** this case (if not, you have infinite recursion)
- A recursive step
  - The **same problem**
  - Just **slightly different** in a way that will eventually reach the base case