CSC148 Week 5

Larry Zhang

Announcements

- Test 1 on Friday, Feb 9th
- We will start at 5:00 PM
- The length of the test is 50 minutes
- Closed-book, closed-notes, no calculator.
- The test will have an API sheet same as last year’s test.
- Coverage: everything prior to the test: lectures, handouts, labs, Exercise 1.
- Rooms:
  - UTORID A-N: IB-110
  - UTORID O-Z: IB-120
- Bring your T-Card!
- **The lecture on Feb 9 is cancelled.**

Brief Recap: Calling the superclass' constructor

Both of the following ways work.

```
class Person:

    def __init__(self, name):
        self.name = name

class Student(Person):

    def __init__(self, name, student_num):

        Person.__init__(self, name)
        self.student_num = student_num
```

```
class Person:

    def __init__(self, name):
        self.name = name

class Student(Person):

    def __init__(self, name, student_num):

        super().__init__(name)
        self.student_num = student_num
```

Tracing Recursive Functions

Tracing Recursion: Bottom-Up

When tracing a given recursive function (evaluating the output of a given input), we recommend the following approach.

- start tracing from the simplest cases (no recursive call)
- Trace the next-most complex case by **plugging in known results** of simpler cases
- Keep doing this until you trace the required (most-complex) case

Practice: trace recursion bottom-up

```
def rec_max(lst):
  '''(list of int) -> int
  Return max number in possibly nested list of numbers.

  >>> rec_max([17, 21, 0])
  21
  >>> rec_max([17, [21, 24], 0])
  24
  '''
  nums = []
  for element in lst:
    if isinstance(element, int):
      nums.append(element)
    else:
      nums.append(rec_max(element))
  return max(nums)
```

DEMO:

`rec_max.py`

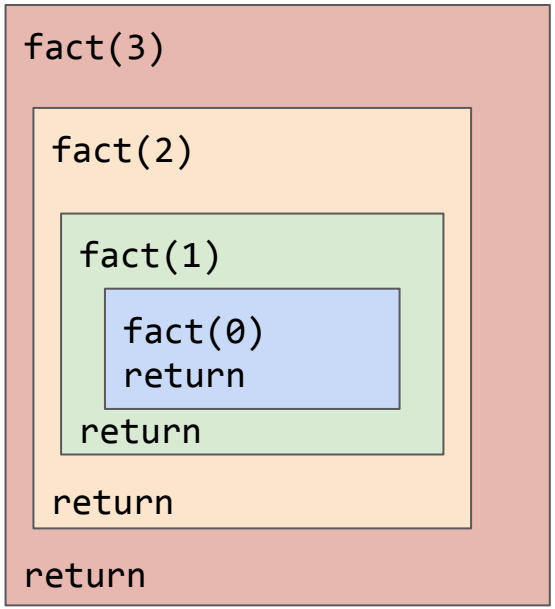
Tracing Recursion **Top-Down**

- The bottom-up approach is convenient for tracing, but it is not how Python (or other languages) actually runs a recursive program.
- The execution starts from the top-level function call, then makes deeper levels of recursive calls.
- In general, when function A calls function B, it's like A go to sleep, B wakes up, finish the task, then return to A, then A wakes up and continue.
- The values needed by the sleeping people is remembered some region in memory.
    - Note the order when A calls B, then B calls C
    - Last Sleep, First Wake Up
- This region is called "**stack**".

Example: factorial

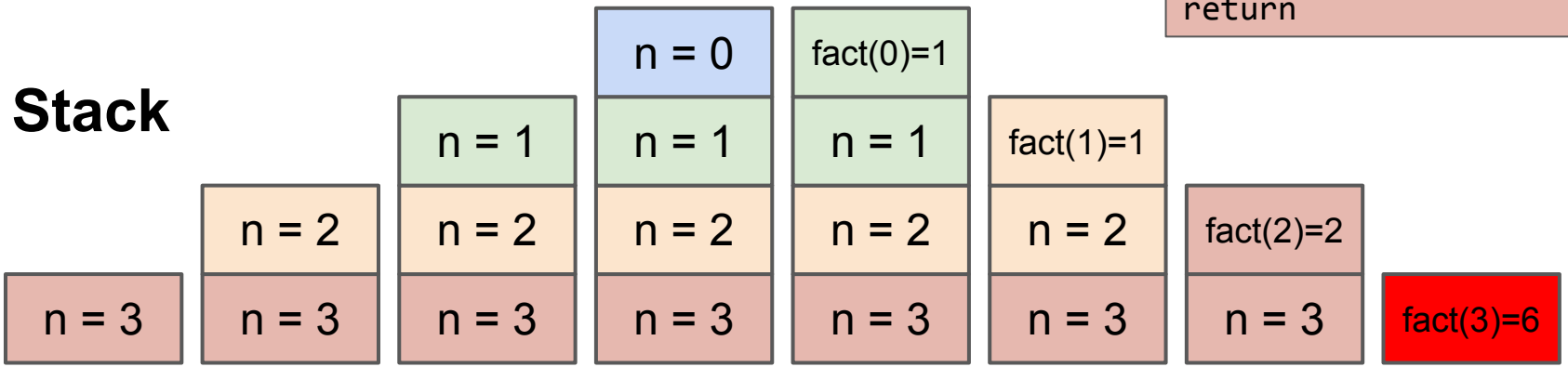
```
def factorial(n):

    if n == 0:
        return 1
    return n * factorial(n-1)
```



Trace: factorial(3)

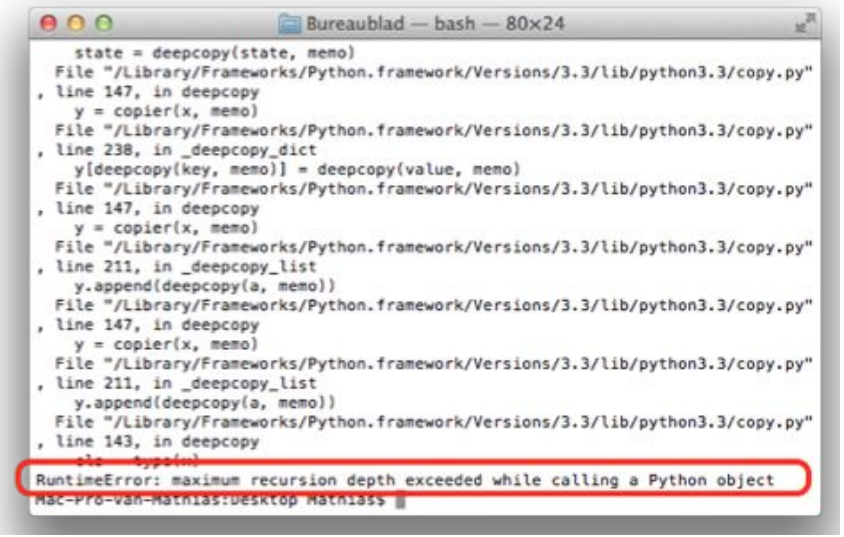
Stack



More levels of recursion ⇒ larger sized stack ⇒ more memory used

Too many levels of recursion ⇒ Exceeding memory usage limit

⇒ RecursionError: maximum recursion depth exceeded (**Stack Overflow**)



Practice: trace recursion bottom-up

```
def rec_max(lst):
  '''(list of int) -> int
  Return max number in possibly nested list of numbers.

  >>> rec_max([17, 21, 0])
  21
  >>> rec_max([17, [21, 24], 0])
  24
  '''
  nums = []
  for element in lst:
    if isinstance(element, int):
      nums.append(element)
    else:
      nums.append(rec_max(element))
  return max(nums)
```

DEMO:

`rec_max.py`

Takeaway

- Tracing bottom-up is convenient, but is not how the flow of execution works in real life.
- Tracing top-down is close to the real flow of execution, but it is less convenient to perform than bottom-up and is also more prone to mistakes (you need to maintain a stack of information while tracing).

Game: Guess the Number

Let's say I pick a number randomly from 1 to 100, and you want to guess it.

I will tell you "too low" or "too high" or "correct".

How many guesses do you need at most to guess the correct number?

- Strategy 1: guess 1, 2, 3, 4, …, until correct
  - worst-case: 100 guesses (if the number is 100)
- Strategy 2: eliminate **half** the possible numbers by guessing the **midpoint** in the range of remaining possibilities
  - worst-case: 50 → 25 → 13 → 7 → 4 → 2 → 1
  - $\lfloor \log_2(n) \rfloor + 1$ = 7 guesses for n = 100

A similar problem

Given a **sorted** array of integers of size n,

when want check if a given number **x** exists in the array.

```
[22, 33, 45, 63, 65, 71, 77, 81, 99, 103, 888, 9990, 121121]
```

- Similar to the game, we can eliminate about half of the list on each iteration
- The strategy is to find the midpoint of the current list.
- Based on this comparison, we know that our item can exist in only one of the two halves, so we **recursively** search there.
- This yields an O(log n) algorithm, where n is the length of the list to search
- This algorithm is called **Binary Search**.

DEMO:
implement binary search

`binary_search.py`

Home exercise

The code we just wrote is very efficient.

Whenever we do slicing like **lst[:mid_index]**, we are **copying** the list, which is time-consuming.

To avoid copying, we can implement a helper function like below:

```
binary_search_helper(lst, value, start_index, end_index)
```

We pass various start and end indices to do binary search in different segments, rather than passing copies of the list.

Implement this yourself.