02-23-2022

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

11: More recursion

David Kauchak

he/him/his

Lectures

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

Labs

Lecture 11: More recursion

- Recursion

Writing recursive functions

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
    - Pretend you had a working version of your function, but it only works on smaller versions of your current problem.
        - The recursive problem should be getting "smaller", by some definition of smaller.
            - E.g., for smaller numbers (like in factorial), lists that are smaller/shorter, strings that are shorter
    - other ideas:
        - Sometimes, define it in English first and then translate that into code.
        - Often, nice to think about it mathematically, using equals.
3. Define the base case
    - What is the smallest (or simplest) problem? This is often the base case
4. Put it all together
    - first, check the base case
        - return something (or do something) for the recursive case
    - if the base case isn't true
        - calculate the problem using the recursive definition
        - return the answer

Recursion is similar to induction in mathematics

- Proof by induction in mathematics:
  - 1. show something works the first time (base case).
  - 2. assume that it works for some time.
  - 3. show it will work for the next time (i.e. time after "some time”).
  - 4. therefore, it must work for all the times.

Practice Time

- Write a recursive function called `rec_sum` that takes a positive number as a parameter and calculates the sum of the numbers from 1 up to and including that number.
  - 1. Define what the header function is:
    - `def rec_sum(n)`
  - 2. Define the recursive case:
    - $\sum_{i=1}^{n} = 1 + 2 + 3 + \ldots + (n-1) + n = ???$
      - Can you rewrite this expression so that there's a sum on the right hand side (that's smaller?)
      - Another way to think about it: pretend like we have a function called `rec_sum` that we can use but only on smaller numbers
        - `rec_sum(n) = ?????? rec_sum(?)`
      - `rec_sum(n) = n + rec_sum(n-1)`
        - i.e. the sum of the numbers 1 through `n`, is n plus the sum of the numbers 1 through `n-1`

Practice Time (cont'd)

- Write a recursive function called `rec_sum` that takes a positive number as a parameter and calculates the sum of the numbers from 1 up to and including that number.
  - 3. Define the base case:
    - in each case, the number is getting smaller. What's the smallest number we would ever want to have the sum of?
      - 0. What's the answer when it's 0? 0!
  - 4. put it all together! - look at the `rec_sum` function in `recursion.py` code
    - Check the base case first:
      - `if n == 0`
    - Otherwise:
      - Do exactly our recursive relationship

Practice Time

- Write a recursive function called `rec_sum_list` that takes a list of numbers as a parameter and calculates their sum.
  - 1. Define what the function header is:
    - `def rec_sum_list(some_list)`
  - 2. Define the recursive case:
    - Pretend like we have a function called rec_sum_list that we can use but only on smaller lists
      - what would we get back if we called rec_sum_list on everything except the first element?
        - the sum of all of those elements
      - how would we get the sum to the entire list?
        - just add that element to the sum of the rest of the elements
    - The recursive relationship is:
      - `rec_sum_list(some_list) = some_list[0] + rec_sum_list(some_list[1:])`

Practice Time (cont'd)

- Write a recursive function called `rec_sum_list` that takes a list of numbers as a parameter and calculates their sum.
  - 3. Define the base case:
    - in each case, the list is getting smaller.
    - Eventually, it will be an empty list. What is the sum of an empty list?
      - 0.
  - 4. put it all together! - look at the `rec_sum_list` function in `recursion.py` code
    - Check the base case first:
      - `if some_list == []`
      - Could have also written `if len(some_list) == 0`
    - Otherwise:
      - Do exactly our recursive relationship

Practice Time (cont'd)

- What does this work? Let's look at an example
  - `rec_sum_list([1, 2, 3, 4])`
    - `1 + rec_sum_list([2, 3, 4])`
      - `2 + rec_sum_list([3, 4])`
        - `3 + rec_sum_list([4])`
          - `4 + rec_sum_list([])`
          - `4 + 0`
        - `3 + 4`
      - `2 + 7`
    - `1 + 9`
  - `10`
- Look at `rec_sum_list_print` in [`recursion.py`](recursion.py) to see how `print` statements reveal the recursion.

Practice Time

- Write a recursive function called `reverse` that takes a string as a parameter and reverses the string.
  - 1. Define what the function header is:
    - `def reverse(some_string)`
  - 2. Define the recursive case:
      - Pretend like we have a function called `reverse` that we can use but only on smaller strings
        - To reverse a string:
          - remove the first character,
          - reverse the remaining characters,
          - put that first character at the end
    - The recursive relationship is:
      - `reverse(some_string) = reverse(some_string[1:]) + some_string[0]`

Practice Time (cont'd)

- Write a recursive function called `reverse` that takes a string as a parameter and reverses the string
  - 3. Define the base case:
    - in each case, the string is getting shorter.
    - Eventually, it will be an empty string. What is the reverse of an empty string?
      - `""`
  - 4. put it all together! - look at the `reverse` function in `recursion.py` code
    - Check the base case first:
      - `if some_string == ""`
      - Could have also written `if len(some_string) == 0`
    - Otherwise:
      - Do exactly our recursive relationship
- Look at `reverse_print` in [`recursion.py`] to see how `print` statements reveal the recursion.

Practice Time

- Write a recursive function called `power` that takes a `base` and an `exponent` as parameters and returns $base^{exponent}$.
  - That is it calculates `base**exponent` without using the `**` operator. You can assume a positive exponent.
  - 1. Define what the function header is:
    - `def power(base, exponent)`
  - 2. Define the recursive case:
    - $base^{exponent} = base^{exponent-1} * base$

Practice Time (cont'd)

- Write a recursive function called `power` that takes a `base` and an `exponent` as parameters and returns $base^{exponent}$.
  - 3. Define the base case:
    - in each case, the exponent is getting smaller.
    - Eventually, the exponent will be 0.
      - $base^0 = 1$
  - 4. put it all together! - look at the `power` function in `recursion.py` code
    - Check the base case first:
      - `if exponent == 0`
    - Otherwise:
      - Do exactly our recursive relationship.

Practice Time

- What does `rec_mystery` function in [`mystery_recursion.py`] do?
    - Recursive function.
    - Work through a small example, e.g., `rec_mystery([2, 4, 3, 1])`
        - `rec_mystery([2, 4, 3, 1]) # compares m = 4 and l[0] = 2 and returns 4`
            - `rec_mystery([4, 3, 1]) # compares m = 3 and l[0] = 4 and returns 4`
                - `rec_mystery([3, 1]) # compares m = 1 and l[0] = 3 and returns 3`
                    - `rec_mystery([1]) # returns 1`
    - Returns the maximum element in the list!

Practice Time (cont'd)

- Returns the maximum element in the list! How?
  - 1. `rec_max(l)`
  - 2. `rec_max(l) = ??? rec_max(l[1:])`
    - assume/trust that the recursive call works
    - if it does, then it will return the largest value in `l[1:]`
    - the largest value of the whole list is then either the first element (`l[0]`) or the largest value in the rest of the list (`rec_max(l[1:])`)
  - 3. The list will get smaller and smaller. `max([])` doesn't really make sense, so our base case will be when there's a single element.
  - Recursive case:
    - make a recursive call on the rest of the list
    - store that value in `m`
    - compare `m` to the first element and return whichever is larger

Practice Time

- Look at the `spiral` function in [`turtle_recursion.py`] do?
    - what would the picture look like if I called `spiral(80, 50)`
        - What does this function do?
            - Draws a spiral on the screen recursively.
                - `forward 80`
                - `left 30`
                - `spiral( 76, 49 )`
                    - `forward 76`
                    - `left 30`
                        - `spiral(72.2, 48)`
                            - `forward 72.2`
                            - `left 30`

Practice Time (cont'd)

- When does it stop?
  - When levels = 0.
    - We put a dot there to make it explicit.
- Repeat 50 times:
  - forward length
  - left 30
  - reduce length by 5%

Practice Time (cont’d)

- What if we wanted to end up back at the starting point, but we couldn't pick the pen up? We could trace our steps backwards.
    - Assume that the recursive call returns back to its starting point. What would we need to do to make sure that our call returned back to the starting point?
    - Add the following after the recursive call:
        - `right(30)`
        - `backward(length)`
    - if we run it now, we draw the spiral all the way down, and then we retrace backwards.:
        - each call to spiral retraces its own part after the recursive call.
        - the stack keeps track of each of the recursive calls.

Practice Time

- Run the `broccoli_demo` function in [`turtle_recursion.py`]
  - 1. Define what the header function is:
    - `broccoli(x, y, length, angle)`
  - 2. Define the recursive case:
    - broccoli is a line with three other broccolis at the end:
      - one directly straight out
      - one 20 degrees to the left
      - one 20 degrees to the right
    - the three other broccolis should be smaller/shorter than the current

Practice Time (cont'd)

- Run the `broccoli_demo` function in [`turtle_recursion.py`]
    - 3. Define the base case:
        - in each case, the length of the broccoli to be drawn gets shorter.
        - We stop at length < 10 and place a yellow dot
    - 4. put it all together! - look at the `power` function in `recursion.py` code
        - Check the base case first:
            - `if length < 10`
                - Draw a yellow dot.
        - Otherwise:
            - draw three smaller broccolis at different angles.
- new_x and new_y are the ending coordinates of the line being drawn. We save them because after the first recursive call to broccoli the turtle won't be in the same place.

Resources

- Textbook: Chapter 16
- `recursion.py`
- `mystery_recursion.py`
- `turtle_recursion.py`

Practice Problems

- Practice 8 (solutions)

Homework

- Assignment 5 (ongoing)