CIS 1100

Recursion

Python
Fall 2024
University of Pennsylvania

Learning Objectives

- To understand how to think recursively
- To be able to write recursive functions
- To be able to trace a recursive function
- To be able to write recursive algorithms and functions for searching arrays

CIS 1100

Thinking Recursively

Python
Fall 2024
University of Pennsylvania

Recursive Thinking

The journey of a thousand miles starts with one mile.

And then a journey of 999 miles.

Recursive Thinking

A function is recursive if it invokes itself to do part of its work.

Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means.

Recursion reduces a problem into one or more simpler versions of itself.

Recursion

An alternate to using loops for solving problems

The core of recursion is taking a big task and breaking it up into a series of related small tasks.

- Example: handing out papers for an exam
  - Iterative: have a TA walk down a row of students, giving each person an exam
  - Recursive: A student takes one exam, pass the rest down the aisle

Breaking up a large problem

We want to write a program that prints N stars on one line, but without loops.

```
def print_stars(N):
```

Here's

```
print_stars(N) ---> print_stars(1) + print_stars(N - 1)
print_stars(3) ---> print_stars(1) + print_stars(2)
print_stars(2) ---> print_stars(1) + print_stars(1)
print_stars(1) ---> print("*");
```

CIS 1100

Mechanics of Recursion

Python
Fall 2024
University of Pennsylvania

Anatomy of a Recursive Function

Every recursive function needs at least one **base case** and at least one **recursive part**.

The **base case:**

- handles a simple input that can be solved without resorting to a recursive call. Can also be thought of as the case where we "end“ our recursion.

The **recursive part:**

- contains one or more recursive calls to the function.
- In every recursive call, the parameters must be in some sense "closer" to the base case than those of the original call

Anatomy of a Recursive Function

Pretend that you never learned about `*` as an operator!

```
def multiply(x, y):
    """Function takes two ints x and y and returns x times y
    Inputs:
        x, the first operand
        y, the second operand
    Returns:
        x * y
    """
    if x == 1:
        return y                          # base case
    else:
        return y + multiply(x - 1, y)     # recursive call
```

Run-Time Stack and Activation Frames

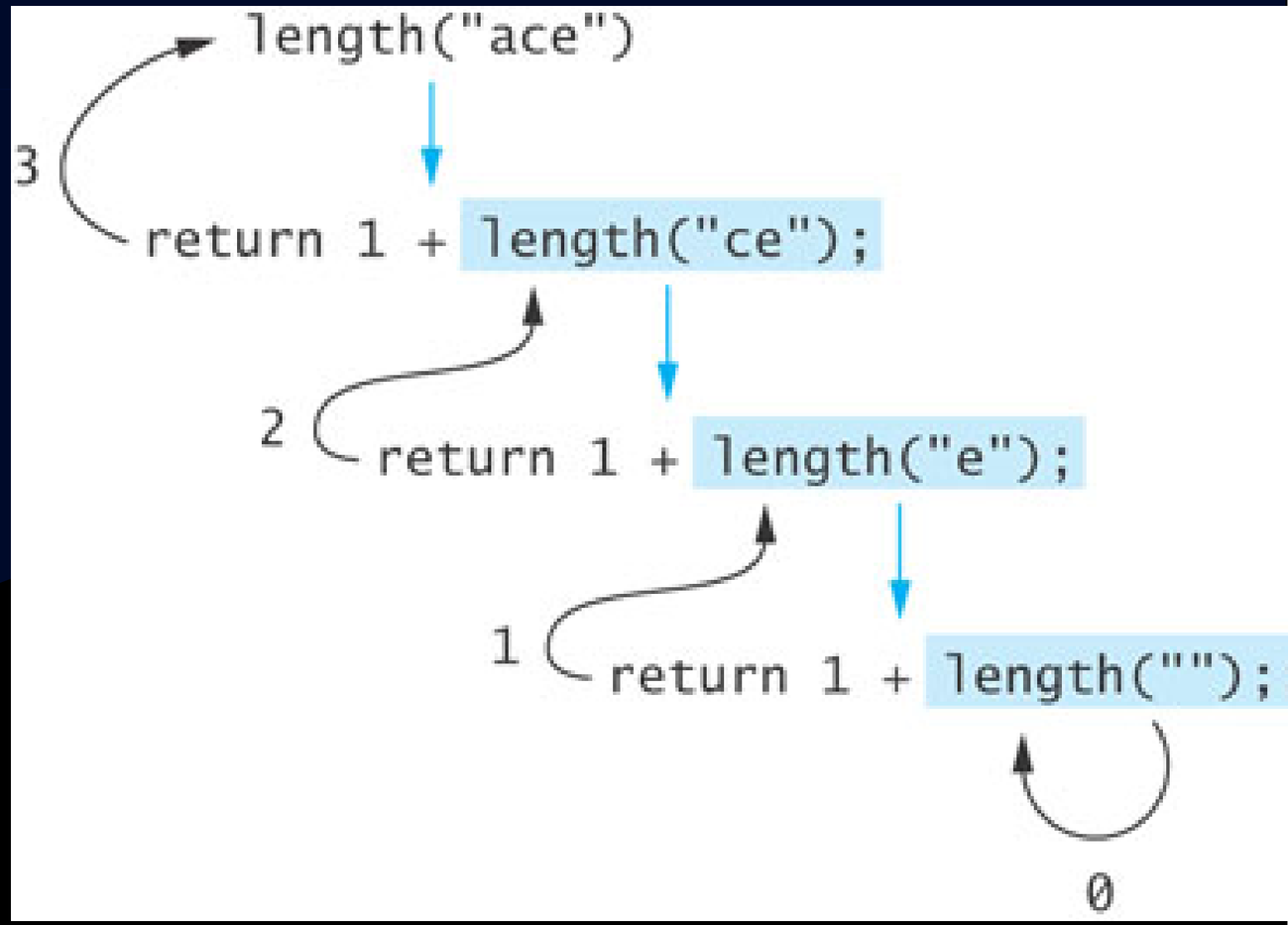
Python maintains a run-time stack on which it saves new information in the form of an *activation frame*, which stores:

- function arguments
- local variables (if any)
- the return address of the instruction that called the function

Whenever a new function is called (recursive or not), Python pushes a new activation frame onto the run-time stack

Tracing a Recursive Function

The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + multiply(1, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + multiply(1, 10)`	pending
`multiply(1, 10)`	`1`	`10`	`10`	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + 20` → `30`	complete
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + 30` → `40`	complete
`multiply(3, 10)`	`3`	`10`	`10 + 20` → `30`	complete
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

So `multiply(4, 10)` evaluates to `40`.

CIS 1100

How To Write
Recursive Functions

Python
Fall 2024
University of Pennsylvania

Steps to Design a Recursive Algorithm

Identify the base case(s) and solve it/them directly

- There must be at least one case (the base case),
  for a small value of *n*, that can be solved directly
  Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- A problem of a given size *n* can be reduced to one or more
  smaller versions of the same problem (recursive case(s))
  Combine the solutions to the smaller problems to solve the larger problem

Design a Recursive Solution to a Problem

```
def is_palindrome(s):
```

Given a string, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?**
- **Recursive steps?**

Design a Recursive Solution to a Problem

```
def is_palindrome(s):
```

Given a string, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?** Strings of length 0 or 1 are palindromes always! (`""` and `"a"`, for example))
- **Recursive steps?**

Design a Recursive Solution to a Problem

```
def is_palindrome(s):
```

Given a string, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?** Strings of length 0 or 1 are palindromes always! (`""` and `"a"`, for example))
- **Recursive steps?** For any string, it can only be a palindrome if its first and last letters are the same.
  - If they are, then we still need to check the rest of the string
  - If they are not the same, then we can stop immediately.

Design a Recursive Solution to a Problem

```
def is_palindrome(s):
    if ???:
        return ???
    else:
        return ???
```

Design a Recursive Solution to a Problem

What's the condition for the base case?

```
def is_palindrome(s):
    if len(s) <= 1:
        return ???
    else:
        return ???
```

Design a Recursive Solution to a Problem

What do we return for a string that's empty or 1 character long?

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    else:
        return ???
```

Design a Recursive Solution to a Problem

How do we check if the first and last characters are the same?

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    else:
        first_last_match = s[0] == s[-1]
        return ???
```

Design a Recursive Solution to a Problem

How do we use this boolean value?

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    else:
        first_last_match = s[0] == s[-1]
        rest = s[1:-1]
        return first_last_match and is_palindrome(rest)
```

Thinking Formulaically

Like writing loops, there's a formula to writing recursive solutions to problems.

1. Identify the kinds of inputs where the solution is very easy to solve.
2. Identify the kinds of inputs where the solution is not easy to solve,
   but can be broken down into smaller versions of the same problem.
3. Figure out how to combine the smaller versions of the problem to solve the larger problem.

Recursion & Lists

Recall that we want to make the problem *smaller*: how to do this with lists?

- Use an index (or indices) to specify the portion of the list that you're recursing over?
  - Good for languages where it's not "easy" to slice lists
  - Technically more memory-efficient
- Make copies of smaller ranges of the list until the problem becomes trivial?
  - Easy in Python, similar to strategy in other recursive-minded languages

Recursion & Lists

```
def find_largest(lst):
```

Given a list of ints, return the largest int in the list **at or after the current index.**

- **Base case?**
  - Empty lists have no largest element, so return `float('-inf')`
  - If a list consists of just one element, that element is the biggest!
- **Recursive steps?**
  - Return whichever is greater: the first element, or the greatest element in the rest of the list.

Recursion & Lists

```
def find_largest(lst):
    if ???:
        return ???
    else:
        return ???
```

Recursion & Lists

```
def find_largest(lst):
    if len(lst) == 0:
        return float('-inf')
    else:
        return ???
```

Recursion & Lists

```
def find_largest(lst):
    if len(lst) == 0:
        return float('-inf')
    else:
        head = lst[0]
        biggest_of_rest = find_largest(lst[1:])
        if head > biggest_of_rest:
            return head
        else:
            return biggest_of_rest
```

CIS 1100

Searching & Recursion

Python
Fall 2024
University of Pennsylvania

Recursive Search

Searching a list can be accomplished using recursion; simplest way to search is **a linear search**

- Examine one element at a time starting with the first element and ending with the last
- Return the index of the list where the element was found, or -1 if absent

Recursive Search

- **Base cases?**
  - Empty list, target can not be found; result is -1
  - First element of the list being searched is target; result is the index of first element
- **Recursive steps?**
  - The recursive step searches the rest of the list, excluding the first element

Algorithm for Recursive Linear Search

(Recall that the **feasible search area** refers to the portion of the list that might still contain our target element...)

- if the feasible search area is empty, the result is `-1`
- else if the first feasible element matches the target, the result is the position of the first element
- else, search the list excluding the first element and return the result

Recursive Linear Search

Since we're looking to return an *index*, we might not want to mess with the list and instead recursively modify the index instead.

```
def linear_search(lst, target, index):
    if index >= len(lst):
        return -1
    elif lst[index] == target:
        return index
    else:
        return linear_search(lst, target, index + 1)
```

(To find an element `target` in `lst`, call `linear_search(lst, target, 0)`)

Recursive Binary Search Algorithm

A binary search can be performed only on an list that has been sorted. Remember: rather than looking at the first element, a binary search compares the **middle element** for a match with the target

- **Base cases?**
  - The list is empty
  - Element being examined matches the target
- **Recursive steps?**
  - If the middle element does not match the target, a binary search excludes the half of the list within which the target cannot be found

Design of a Binary Search Algorithm

- if the list is empty, return –1 as the search result
- else if the middle element matches the target, return the subscript of the middle element as the result
- else if the target is less than the middle element, recursively search the list elements before the middle element and return the result
- else recursively search the list elements after the middle element and return the result

Recursive Binary Search

```
def binary_search(lst, target, left, right):
    if left > right or len(lst) == 0:
        return -1
    middle = (left + right) // 2
    if lst[middle] == target:
        return middle
    elif target < lst[middle]:
        return binary_search(lst, target, left, middle - 1)
    else:
        return binary_search(lst, target, middle + 1, right)
```

(To find an element `target` in `lst`, call `binary_search(lst, target, 0, len(lst))`)

Recursive Helper Functions

Kind of annoying to have to remember how to "set up"

the `binary_search` call each time you want to use it!

- rename the previous function to e.g. `binary_search_helper`
- write another function with the name `binary_search`
  that calls the other but sets up the indices for you:

```
def binary_search(lst, target):
    return binary_search_helper(lst, target, 0, len(lst))
```