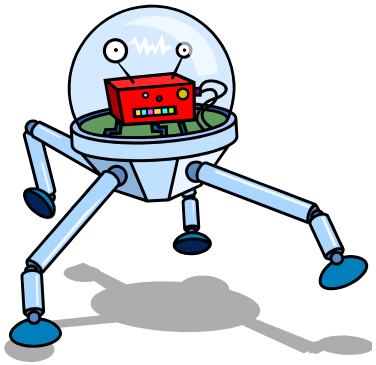
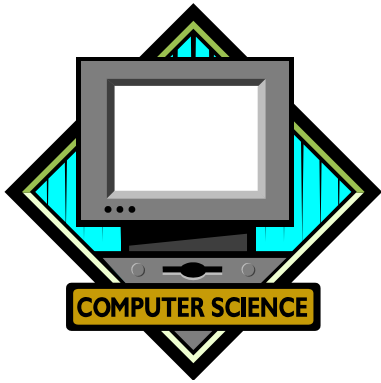
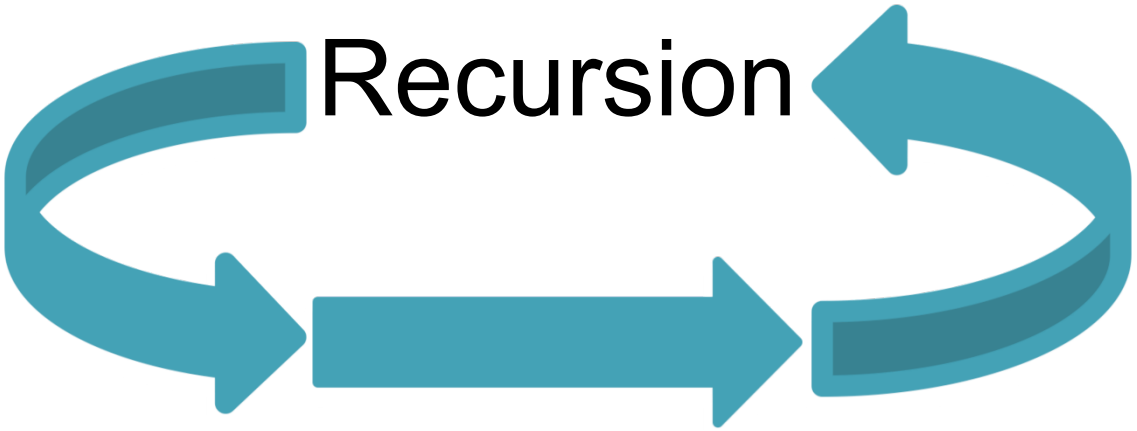
Caltech/LEAD Summer 2012 Computer Science





Lecture 7: July 13, 2012

Recursion



This lecture

- Minor stuff
  - Boolean operators: `not`, `and`, `or`
  - `pass`
  - sequence slices
- Recursion
  - what it is
  - how it works
  - examples

Boolean operators

- A "boolean operator" is an operator that returns a boolean value (`True` or `False`)
- Unlike numeric operators (like `+ - * /`) boolean operators are words, not symbols
- The three boolean operators are `not`, `and`, `or`
- They are usually used in `if` statements to combine or modify the test part of the `if` statement

not

- The `not` operator negates a boolean value (`True` becomes `False`, `False` becomes `True`)

```
>>> not (10 == 10)
False
>>> not (5 > 6)
True
```

and

- The **and** operator combines two boolean expressions, returning **True** if both of the expressions return **True**

```
>>> (10 == 10) and (5 < 6)
True
>>> a = 10
>>> if a > 5 and a < 15:
...     print 'a is between 5 and 15'
a is between 5 and 15
```

or

- The `or` operator combines two boolean expressions, returning `True` if *either* of the expressions return `True`

```
>>> (10 == 10) or (5 > 6)
True
>>> a = 10
>>> if a < -5 or a > 5:
...     print 'abs(a) > 5'
abs(a) > 5
```

Precedence

- `not` has a higher precedence than `and` or `or`
- We can write

```
(a > 5) and not (a > 10)
```

- and it means the same as:

```
(a > 5) and (not (a > 10))
```

pass

- Sometimes we have the body of a loop or an `if` statement that we wish to be empty (at least for now)
- Usually this means that we are going to put some code in it later, but we haven't gotten around to it yet
- If we don't have anything in the body, it would be a syntax error
- We can use the `pass` statement for this

pass

- Example: This:

```
if a > 10:
# nothing here, fill in later
```

- is a syntax error:

```
IndentationError: expected an indented block
```

- You can write it like this instead:

```
if a > 10:
    pass
# do nothing for now, fill in later
```

pass

- `pass` works with `if` statements, `elif` and `else` blocks, `for` loops, `while` loops, and even function bodies!
- There is rarely a need to leave a `pass` statement in completed code
- Usually easy to re-write the code to eliminate the `pass` statement

pass

- Example:

```
if a >= 5:
    pass
else:
    print 'a is less than 5'
```

- Can rewrite this as:

```
if a < 5:
    print 'a is less than 5'
```

Sequence slices

- Python allows you to get a single element from a sequence using the square bracket notation:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2]
3
```

- This works for all sequences (tuples, strings), not just lists

Sequence slices

- Python also lets you get more than one element from a sequence using a slice:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2:4]  # a slice of a list
[3, 4]
```

- Again, this works for all sequences

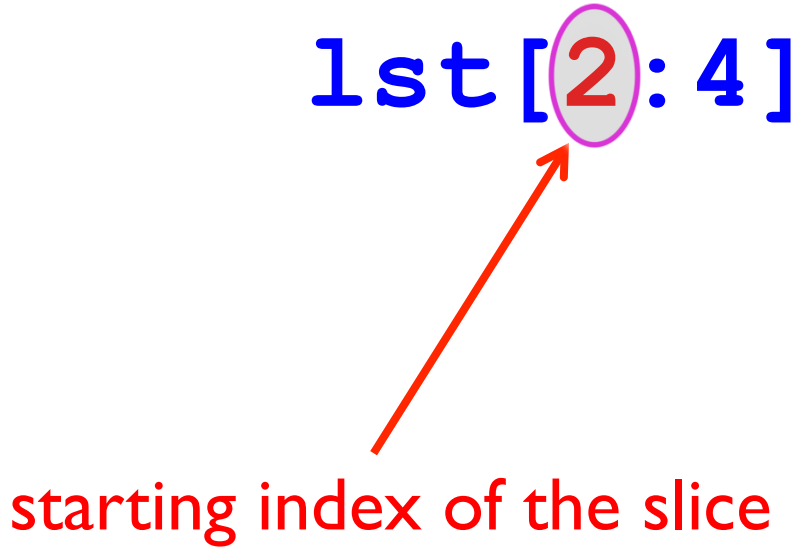
Slice notation

- Anatomy of a slice:

```
lst[2:4]
```

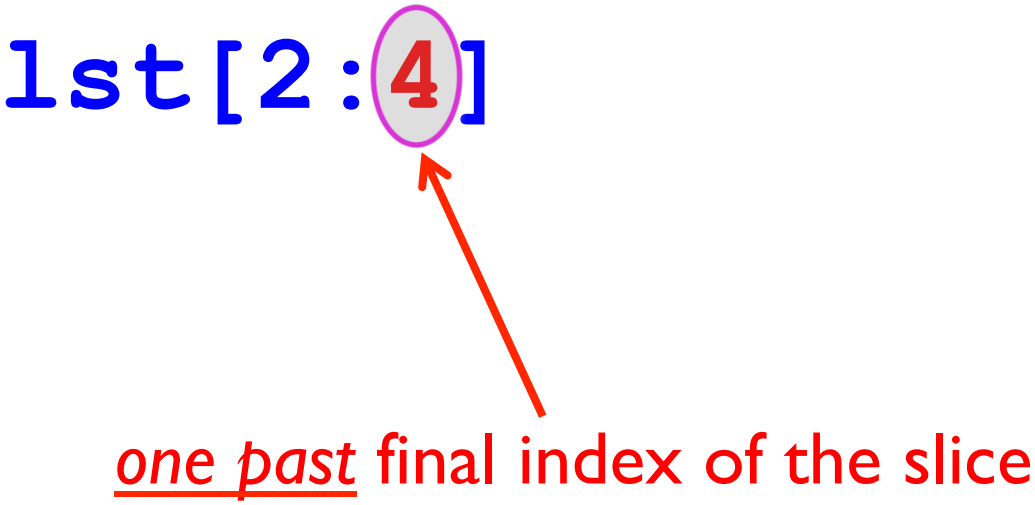
Slice notation

- Anatomy of a slice:



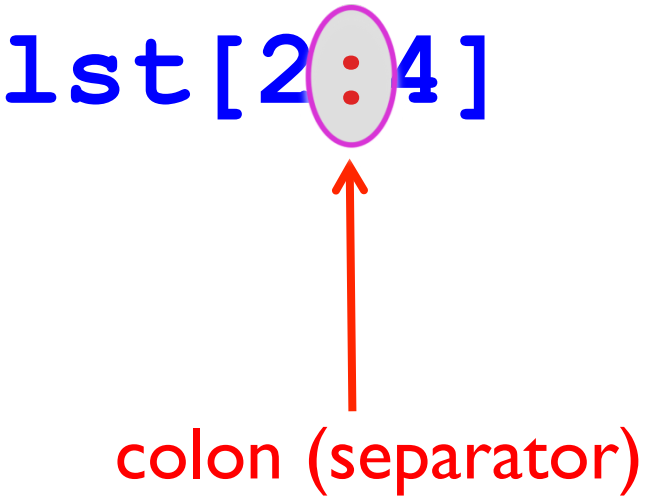
Slice notation

- Anatomy of a slice:



Slice notation

- Anatomy of a slice:



Sequence slices

- A sequence slice makes a *copy* of a chunk of the sequence
- First element of the copy is the element of the original sequence at the starting index of the slice
- Last element of the copy is the element of the original sequence *one before* the final index of the slice

Sequence slices

- Examples:

```
>>> lst = [1, 2, 3, 4, 5]
>>> tup = (6, 7, 8, 9, 10)
>>> s = 'abcdef'
>>> lst[0:5]
[1, 2, 3, 4, 5]
>>> tup[1:5]
(7, 8, 9, 10)
>>> s[1:4]
'bcd'
```

Sequence slices

- If the slice's final index is larger than the length of the sequence, the slice ends at the last element

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[3:1000]
[4, 5]
```

- If the slice's final index is left out, it's assumed to be equal to the length of the sequence

```
>>> lst[3:]
[4, 5]
```

Sequence slices

- If the slice's starting index is left out, the slice starts at the first element

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[0:3]
[1, 2, 3]
>>> lst[:3]
[1, 2, 3]
```

Sequence slices

- If *both* the slice's starting index and the slice's ending index are left out, the slice is a copy of the entire list!

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[0:5]
[1, 2, 3, 4, 5]
>>> lst[:]
[1, 2, 3, 4, 5]
```

Sequence slices

- Slices can use negative indices (counting from the end of the sequence)

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[:-1]  # -1: last element of list
[1, 2, 3, 4]
>>> lst[:-2]
[1, 2, 3]
>>> lst[-3:-1]
[3, 4]
```

Sequence slices

- Common application: remove newline character from the end of a string:

```
>>> s = 'Hello, world!\n'
>>> s[:-1]
'Hello, world!'
```

Interlude

- Some demo programs!

Recursion

- Recursion is when you have a function that calls itself
- Recursion is very useful when you have a problem whose solution can be defined in terms of a smaller version of the problem
- Recursion can also be thought of as a different way to write a loop

Recursion

- Simple example: the `factorial` function
- You defined it in lab 1
- You probably used a `for` loop or maybe a `while` loop
- Now we'll show a different way to define it, without using `for` or `while`

factorial

```
def factorial(n):
    '''Compute the factorial of n, recursively.'''
    if n == 0:
        return 1
    else:
        return (n * factorial(n - 1))
```

- Notice: **`factorial`** is defined in terms of itself!
- This reflects the mathematical definition:

factorial(0) = 1

factorial(n) = n * factorial(n – 1)

`factorial`

- It might seem weird that you can use `factorial` in its own definition
- Doesn't cause problems: once the body of `factorial` is evaluated, `factorial` itself will already have been defined, so everything works
- Let's evaluate `factorial(5)` to see how this works

factorial

```
factorial(5)
```

- This becomes:

```
if 5 == 0:
    return 1
else:
    return (5 * factorial(5 - 1))
```

- which simplifies to:

```
5 * factorial(4)
```

factorial

```
5 * factorial(4)
```

- This becomes:

```
5 * (if 4 == 0:
        return 1
    else:
        return (4 * factorial(4 - 1)))
```

- which simplifies to:

```
5 * 4 * factorial(3)
```

factorial

```
5 * 4 * factorial(3)
```

- This becomes:

```
5 * 4 * (if 3 == 0:
             return 1
         else:
             return (3 * factorial(3 - 1)))
```

- which simplifies to:

```
5 * 4 * 3 * factorial(2)
```

factorial

```
5 * 4 * 3 * factorial(2)
```

- This becomes:

```
5 * 4 * 3 *
   (if 2 == 0:
        return 1
    else:
        return (2 * factorial(2 - 1)))
```

- which simplifies to:

```
5 * 4 * 3 * 2 * factorial(1)
```

factorial

```
5 * 4 * 3 * 2 * factorial(1)
```

- This becomes:

```
5 * 4 * 3 * 2 *
  (if 1 == 0:
       return 1
   else:
       return (1 * factorial(1 - 1)))
```

- which simplifies to:

```
5 * 4 * 3 * 2 * 1 * factorial(0)
```

factorial

```
5 * 4 * 3 * 2 * 1 * factorial(0)
```

- This becomes:

```
5 * 4 * 3 * 2 * 1 *
   (if 0 == 0:
        return 1
    else:
        return (0 * factorial(0 - 1)))
```

- which simplifies to:

```
5 * 4 * 3 * 2 * 1 * 1
```

factorial

```
5 * 4 * 3 * 2 * 1 * 1
```

- This becomes:

```
120
```

- which is the answer!
- This is tedious for us to compute, but not for the computer

factorial

- Let's look at the definition again:

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

- What happens if we leave out the `n == 0` case?

factorial

- We would have:

```
def factorial(n):
    return (n * factorial(n - 1))
```

- This would never terminate!
- Recursively-defined functions like factorial need to have a *base case* which can immediately return a value (without a recursive function call)
- Here, the `n == 0` case is the base case

factorial

```
def factorial(n):
    if n == 0:
        return 1    # base case
    else:
        return (n * factorial(n - 1))
```

- The base case can be computed immediately, without any recursive calls to **`factorial`**
- It stops the chain of recursive calls to **`factorial`**, which allows the function to terminate
- All recursive functions need to have one or more base cases!

reverse

- That was a pretty simple example
- Let's try a more complicated example: reversing a list
- Python has a built-in method called `reverse` for lists, but that reverses a list in-place
- We'll define a function that returns the reverse of a list without altering the original list

reverse

- Let's start by thinking about how to solve the problem in terms of a smaller version of itself
- Assume that we are reversing a list called `lst`
- Assume that we know what the reverse of the slice of the list starting at index 1 is (`lst[1:]`)
- How can we use this to compute the reverse of the entire list `lst`?

reverse

```
>>> lst = [1, 2, 3, 4, 5]
>>> rev_lst1 = [5, 4, 3, 2]
>>> rev_lst = rev_lst1 + [1]
>>> rev_lst
[5, 4, 3, 2, 1]
```

- This is the answer we wanted (the reverse of `lst`)
- Let's use this to write the **`reverse`** function

reverse

```
def reverse(lst):
    '''Return the reverse of a list.'''
    return (reverse(lst[1:]) + [lst[0]])
```

- This is the right idea
- However, this won't work properly – why not?
- Let's try evaluating `reverse([1,2,3])` and see what happens

reverse

```
def reverse(lst):
    '''Return the reverse of a list.'''
    return (reverse(lst[1:]) + [lst[0]])
```

- `reverse([1, 2, 3])`
- `reverse([2, 3]) + [1]`
- `(reverse([3]) + [2]) + [1]`
- `((reverse([]) + [3]) + [2]) + [1]`
- Error! (`[]` has no element at index `0`)
- Problem: We forgot the base case!

`reverse`

- The base case is the argument that can be computed without having to make a recursive call
- Here, we can use the empty list (`[]`) as the base case
- We want `reverse([])` to be equal to `[]`
- Let's re-write the code accordingly

reverse

```
def reverse(lst):
    '''Return the reverse of a list.'''
    if lst == []:   # base case
        return []
    else:
        return (reverse(lst[1:]) + [lst[0]])
```

- Now it works!
- Time for one last example

sort

- A common problem is how to sort a list of items (say, numbers)
- Again, Python has a method for this:

```
>>> lst = [5, 1, 3, 2, 4]
>>> lst.sort()
>>> lst
[1, 2, 3, 4, 5]
```

- Again, this changes the list in-place

sort

- We might want to define a function called `sort` which returns a sorted version of a list without changing the original list
- There are lots of different ways (algorithms) to do this
- We'll choose a short and elegant one called "insertion sort"

insert

- We'll assume we have already defined a function called `insert` which will insert a number into an already sorted list of numbers, returning a new list

```
>>> insert(3, [])
[3]
>>> insert(3, [1, 2, 4, 5])
[1, 2, 3, 4, 5]
```

`insert`

- We can define **`insert`** using recursion or without using recursion (exercise for the student!), but we won't do that here
- Instead, let's use **`insert`** to define a function to sort lists of numbers
- Basic idea: to sort a list:
  - sort the sublist starting after the first element
  - insert the first element into the sorted sublist
- This is called "insertion sort"

insertion_sort

- First attempt:

```
def insertion_sort(lst):
    '''Sort the list 'lst' using insertion sort.'''
    sorted_rest = insertion_sort(lst[1:])
    return (insert(lst[0], sorted_rest))
```

- This is the right idea, but...
- does this work?

insertion_sort

- First attempt:

```
def insertion_sort(lst):
    '''Sort the list 'lst' using insertion sort.'''
    sorted_rest = insertion_sort(lst[1:])
    return (insert(lst[0], sorted_rest))
```

- This has no base case!
- Eventually, you have the case `lst == []` which cannot be divided further
- We have to handle this case separately

insertion_sort

- Second attempt:

```
def insertion_sort(lst):
    '''Sort the list 'lst' using insertion sort.'''
    if lst == []:    # base case
        return []
    else:
        sorted_rest = insertion_sort(lst[1:])
        return (insert(lst[0], sorted_rest))
```

- This works!

When to use recursion

- Many functions can be written using recursion or without recursion
  - including all the ones we've talked about today!
- In general, you want to use recursion if the recursive definition is significantly easier to write than the non-recursive one
- Recursion is tricky for new programmers, and takes practice
- Upcoming labs will walk you through some examples

Next lectures

- Program design
- Debugging
- Files
- Dictionaries