CS 1
Introduction to Computer Programming

Lecture 23: December 3, 2012
Advanced topics, part 1
Last time

- **Tkinter** widgets and GUI programming
Today

- Advanced topics, lecture 1
  - recursion
  - first-class functions
  - lambda expressions
  - higher-order functions
    - map, filter, reduce
Admin notes

- This is the last week of classes!
- Assignment 7 is due Friday at 2 AM
  - last assignment!
- Make sure you get caught up on all old assignments/redos!
Recursion

- A function is known as *recursive* when it calls itself
- Python allows you to define recursive functions
- Recursion has a reputation for difficulty, but it's really no big deal
- Let's look at a simple example
Sums

- How would you compute the sum of all the numbers from 0 to N (where N >= 0)?
- In Python, could just use the built-in `sum` function or a `for` or `while` loop.
- Mathematically, we can define a solution like this:
  - `sum_to_N(0) = 0`
  - `sum_to_N(n) = n + sum_to_N(n-1)`
Sums

- **sum_to_N** definition:
  - \( \text{sum}_\text{to}_N(0) = 0 \)
  - \( \text{sum}_\text{to}_N(n) = n + \text{sum}_\text{to}_N(n-1) \)

- We state that \( \text{sum}_\text{to}_N(0) \) is 0 by definition
- This is called a *base case*
  - it can be solved *immediately* (no recursion needed)

- To compute \( \text{sum}_\text{to}_N(n) \) for \( n > 0 \):
  - first compute \( \text{sum}_\text{to}_N(n-1) \)
  - then add \( n \) to it
  - This *must* be the correct answer!
Sums

- **sum_to_N** definition:
  - \( \text{sum}_\text{to}_N(0) = 0 \)
  - \( \text{sum}_\text{to}_N(n) = n + \text{sum}_\text{to}_N(n-1) \)

- Another way to look at this:
  - If \( \text{sum}_\text{to}_N(n-1) \) gives the correct sum from 0 to \( n-1 \)
  - Then \( \text{sum}_\text{to}_N(n) \) will be the correct sum from 0 to \( n \), because we've just added \( n \) to \( \text{sum}_\text{to}_N(n-1) \)
Sums

- **sum_to_N** definition:
  - \( \text{sum\_to\_N}(0) = 0 \)
  - \( \text{sum\_to\_N}(n) = n + \text{sum\_to\_N}(n-1) \)

- The hard part:
- **We assume** that \( \text{sum\_to\_N} \) works correctly for all numbers \(< n\) when we define \( \text{sum\_to\_N} \) for the argument \( n \)
- How can we do this?
Sums

- **sum_to_N definition:**
  - `sum_to_N(0) = 0`
  - `sum_to_N(n) = n + sum_to_N(n-1)`

- This is an example of *mathematical induction*:
  - if `sum_to_N` works for some base case (`0`)
  - and we know that if `sum_to_N` works for some value `n-1`, *then* it will also work for the value `n`
  - then we conclude that `sum_to_N` works for all `n` from `0` to infinity!
    - works for `0, 1, 2, 3, ...` as far up as we want to go
def sum_to_N(n):
    if n == 0:
        return 0  # base case
    else:
        return n + sum_to_N(n - 1)
**sum_to_N in Python**

- **Question 1**: How is it possible that you can call a function while defining it?
- **Answer**: In fact, you *don't* call the function while defining it
  - You *never* call any functions while defining a function!
  - Instead, you define it with an internal call to what the function *will be* once it has been completely defined
  - The language implementation can handle the bookkeeping for you
sum_to_N in Python

- **Question 2**: What happens to the computation in progress in `sum_to_N` when a new recursive call to the same function starts?

- **Answer**: Every call to `sum_to_N` gets its own frame on the runtime stack, so when a recursive call to `sum_to_N` starts, a new frame is put on the stack, and the original call to `sum_to_N` still has all its own data in its own frame.
  - recursion wouldn't work without a runtime stack!
A better example: sorting

- Recursion may just seem like a cute trick with no practical value
- But recursion can sometimes allow us to solve problems that would be difficult to solve without it
- Case study: sorting a list of numbers
  - Could use the Python `sort` method, but we want to do it ourselves
  - Recursion will be an essential component of our solution
Quicksort

- We will write a function which is a variation on a sorting algorithm called "quicksort"

- The idea:
  - Take the first number of a list of numbers, called the pivot
  - Divide the rest of the numbers into (a) a list of numbers smaller than the pivot, (b) a list of numbers greater than or equal to the pivot
  - Sort these sublists (recursively) using quicksort again!
  - Combine the sorted lists to get the final sorted list
def quicksort(lst):
    if lst == []:  # base case (no items in list)
        return []  # nothing to sort, return as is
    pivot = lst[0]
    less = []
    greater = []
    for item in lst[1:]:  # examine rest of list
        if item < pivot:
            less.append(item)
        else:
            greater.append(item)
    all = quicksort(less) + [pivot] + quicksort(greater)
    return all
Quicksort

- While writing the `quicksort` function, we assume that the `quicksort` function itself will work on all smaller lists than the list given as an argument.
- We make sure that all the recursive calls to `quicksort` are on lists that are strictly smaller than the list that was given as the argument.
- We make sure that we return the correct value for the base case (empty list, no recursion needed).
- And then our function will work!
Recursion

- Recursion is also very useful when using tree-like data structures
  - e.g. made out of nodes which have a value, a left subtree, and a right subtree
- In this case, functions that act on the tree-like data must first deal with smaller trees
- Whenever a function has to deal with data which is a "smaller version of the original data" (lists → sublists, trees → subtrees), then recursion is useful
Recursion

- You'll learn more about recursion if you take CS 2
  - and *lots* more about recursion if you take CS 4!
- We have to move on to other topics
First-class functions

- We have seen that Python functions can be treated as data
  - e.g. callback functions for event handlers in Tkinter
- This is a general phenomenon
- Python functions are also Python objects, so they can be treated like any other Python object
  - assigned to a different name
  - stored in data structures
  - returned from functions
  - etc.
First-class functions

def double(x):
    return 2 * x

>>> foo = double

>>> foo(10)
20

>>> lst = [foo, foo, foo]

>>> lst[0](42)
84
First-class functions

- You can return functions from functions:

```python
def make_adder(n):
    def add(x):
        return x + n
    return add

>>> add5 = make_adder(5)
>>> add5(10)
15
>>> make_adder(15)(45)
60
```
First-class functions

def make_adder(n):
    def add(x):
        return x + n
    return add

- The \( n \) in \( \text{add} \) is the \( n \) which was the argument to \( \text{make_adder} \)
- We say that the \( n \) from \( \text{make_adder} \) is in scope inside the \( \text{add} \) function
- Even when \( \text{add} \) is returned, the \( n \) that was the argument to \( \text{make_adder} \) is retained
def make_adder(n):
    def add(x):
        return x + n
    return add

- Notice that the `add` function is a very trivial function that is only used in a single place.
- The name `add` is not really important.
- It would be nice if there was a more compact way to create a "function object" without having to give it a name.
In Python, an "anonymous function" can be created using a **lambda** expression.

The syntax:

```
lambda x: <expression involving x>
```

is the same as writing

```
def func(x):
    return <expression involving x>
```

except that the **lambda** expression has no name.
lambda

- Examples:

  `lambda x : x * 2`

- is the same as:

  `def double(x):
      return (x * 2)`
lambda

- Examples:
  
  `lambda x, y: x * x + y * y`

- is the same as:
  
  `def sum_squares(x, y):
    return (x * x + y * y)`
• **make_adder** again:
  ```python
def make_adder(n):
    return (lambda x: x + n)
  ```
• and that's it!
• We use **lambda** expressions primarily when we are defining a function that will only be used once
• **lambda** really only works for one-line functions
  • Python's indentation-based syntax gets in the way for longer functions
As we will see shortly, \texttt{lambda} is often very handy when used with "higher-order" functions that take functions as arguments.

But first...
Interlude

- Another look at recursion!
Higher-order functions

- Python functions can be used as data by just referring to them by name (no arguments) or by creating a **lambda** expression
- What's the use of this?
- In Python, can define functions which take other functions as their arguments
- These are known as "higher-order" functions
- Several useful ones are built-in
The **map** function takes two arguments:
- a function requiring one argument
- a list

and applies the function argument to every element of the list, collecting all the results into a new list

Let's see some examples
def double(x):
    return x * 2

>>> map(double, [1, 3, 5, 7, 9])
[2, 6, 10, 14, 18]

>>> map(lambda x: x * 2, [1, 2, 3, 4, 5])
[2, 4, 6, 8, 10]

- Notice: we can use a lambda expression anywhere we can use a function
\textbf{map} allows us to convert a list into another (related) list of the same size where the elements of the second list are functions of the elements of the first list

\begin{itemize}
  \item \textit{n.b. map} doesn't change the original list
  \item \textbf{map} takes a function as its first argument, so it's a higher-order function
    \begin{itemize}
      \item sounds scary, but it's no big deal
    \end{itemize}
\end{itemize}
Sometimes, we want to pick out certain elements of a list satisfying particular properties

The properties can be represented by a predicate (a function returning a boolean (True/False) value)
filter

- For instance, if we wanted to pick out all positive elements of a list, a predicate could be:
  ```python
def positive(x):
    '''Return True if x is positive.'''
    return (x > 0)
  ```

- This is so simple, we can just write it as:
  ```python
lambda x: x > 0
  ```
There is a higher-order function called `filter` that takes two arguments:

- a function (a predicate i.e. returning a boolean)
- a list

and returns a new list consisting of all of the elements of the original list that satisfied the predicate (for which the predicate returned `True`)

The original list is "filtered" to give the new list
filter

• Examples:

```python
>>> filter(positive, [-3, 1, -4, 1, -5, 9, -2, 6])
[1, 1, 9, 6]

>>> filter(lambda x: x > 0, [5, -3, -8, 9, 7, -9])
[5, 9, 7]

>>> filter(lambda x: x != 0, [1, 0, 0, 2, 0, 0, 0])
[1, 2]

>>> filter(lambda x: x > 5, [4, 1, -2, 0, 3])
[]

>>> filter(lambda x: x > 10, [])
[]
```
Another common thing to want to do is to take a list and collapse it into a single value which is a function of all the elements of the list.

For instance:
- sum all the elements of a list together
- multiply all the elements of a list together
- find the largest/smallest elements of a list

In all cases, we are "reducing" a list into a single value.
reduce

- Example: sum the elements of a list
- If list elements are \([i, j, k, l \ldots]\) we want to compute
  - \((i + j)\)
  - \(((i + j) + k)\)
  - \(((i + j) + k) + l\)
  - \(etc.\) until all the elements of the list are added together
- In each case, we're adding the previous sum to the next element
Python has a built-in function called `reduce` that does this:

```python
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
120
>>> reduce(lambda x, y: max(x, y), [3, 1, 4, 1, 5])
5
```
The first argument to `reduce` must be a function of two arguments.

Let's call it \( f(x, y) \).

Then `reduce(f, [1, 2, 3, 4, 5])` is:

- \( f(f(f(f(1, 2), 3), 4), 5) \)

Notice the nested \( f \)s.

- Second argument to \( f \) is the next element of the list.
- First argument is the result of reducing all previous elements of the list.
reduce

• If we have:
  
  ```python
def f(x, y):
    return x + y
  ```

• Then `reduce(f, [1, 2, 3, 4, 5])` becomes
  
  ```
  ((((1 + 2) + 3) + 4) + 5)
  ```

• which is just the sum of the list

• `reduce` makes it easy to do repetitive computations like this without having to write explicit loops
If we use reduce on an empty list:
```
reduce(lambda x, y: x + y, [])
```
we get:
```
TypeError: reduce() of empty sequence with no initial value
```
reduce takes an extra (optional) argument that (if present) gives a value to use for the empty list
```
>>> reduce(lambda x, y: x + y, [], 0)
0
```
There is a style of programming which uses higher-order functions and lambda expressions a lot.

It's called functional programming.

- In contrast to e.g. object-oriented programming.

Python supports functional programming, but not as well as some other languages.
- E.g. Scheme, Ocaml, Haskell.

CS 4 will cover functional programming in much greater depth (using Scheme and Ocaml).
Next time

- Last lecture in the class!
- Advanced topics, lecture 2
  - command-line arguments
  - list comprehensions
  - iterators
  - generators
- Where to go from here
def sum_to_N(n):
    if n == 0:
        return 0  # base case
    else:
        return (n + sum_to_N(n - 1))
def quicksort(lst):
    if lst == []:  # base case (no items in list)
        return []  # nothing to sort, return as is

    pivot = lst[0]
    less = []
    greater = []

    for item in lst[1:]:  # examine rest of list
        if item < pivot:
            less.append(item)
        else:
            greater.append(item)

    all = quicksort(less) + [pivot] + quicksort(greater)
    return all
def make_adder(n):
    def add(x):
        return x + n
    return add

# Shorter:
def make_adder(n):
    return (lambda x: x + n)