Lecture #7: Recursion (and a data structure)

Announcements:

• A message from the AWE:

  “The Association of Women in EECS is hosting a 61A party this Sunday (2/9) from 1-3PM in the Woz! Come hang out, befriend other girls in 61A and meet AWE members who have taken it before! There will be lots of food, games, and fun!”

• Guerrilla Sections this weekend. Extra, optional sections to practice HOF and Environment Diagrams this weekend. You’ll be expected to work in groups on questions that range from basic to midterm-level. Details will be announced on Piazza.
Data Structures

- To date, we've dealt with numbers and functions for the most part.
- Although one can do just about anything with these, it's not exactly convenient.
- Example: encode a pair of integers as a single integer:
  \[(x, y) \Leftrightarrow 2^x \cdot 3^y\]
- Every \((x, y)\) pair can be encoded, but extracting \(x\) and \(y\) is a chore.
- So Python (like most languages) provides a set of additional data structures for representing collections of values.
Creating Tuples

• To create (construct) a tuple, use a sequence of expressions in parentheses:

  ()       # The tuple with no values
  (1, 2)   # A pair: tuple with two items
  (1, )    # A singleton tuple: use comma to distinguish from (1)
  (1, "Hello", (3, 4)) # Any mix of values possible.

• When unambiguous, the parentheses are unnecessary:

  x = 1, 2, 3       # Same as x = (1,2,3)
  return True, 5    # Same as return (True, 5)
  for i in 1, 2, 3: # Same as for i in (1,2,3):
Selecting from Tuples

- Can compare, print, or **select** values from a tuple; little else.
- Selection is by explicit item number or “unpacking”:

```python
>>> x = (1, 7, 5)
>>> print(x[1], x[2])
7 5
>>> from operator import getitem
>>> print(getitem(x, 1), getitem(x, 2))
7 5
>>> x = (1, (2, 3), 5)
>>> print(len(x))
3
>>> a, b, c = x
>>> print(b, c)
(2, 3) 5
>>> d, (e, f), g = x
>>> print(e, g)
2, 5
>>> x, y = y, x
???
```
More Selection

Selecting subtuples (*slices*) is also possible:

```python
>>> x = (1, 7, 5, 6)
>>> print(x[1:3], x[0:2], x[:2], x[1:4], x[1:], x[1:2])
(7, 5) (1, 7) (1, 7) (7, 5, 6) (7, 5, 6) (7,)
>>> from operator import getitem
>>> print(getitem(x, slice(1,3)), getitem(x, slice(0,2)))
(7, 5) (1, 7)
>>> a, *b, c = x
>>> print(a, b, c)
1 (7, 5) 6
>>> a, *b = x
>>> print(a, b)
1 (7, 5, 6)
```
Multiple Returns

Tuples provide a useful way to return multiple things from a function:

```python
>>> divmod(38, 5)  # Returns (38//5, 38%5)
(7, 3)

>>> def sumprod(x, y):
...     return x+y, x*y
>>> sumprod(3, 5)
(8, 15)
```
Tuple is a Recursive Type

- Tuple is one type of \textit{value}.
- Values thus include integers, booleans, strings, and tuples (among others).
- Tuples are sequences of 0 or more \textit{values}.
- Therefore, the definitions of “value” and “tuple” are \textit{recursive}: they refer to themselves.
- In this case, we’d say that their definitions are \textit{mutually recursive}, since they each refers to the other.
- Recursive data types and recursive algorithms go together.
Example: How Many Numbers?

• Let’s consider a restricted tuple (call it a “numeric pair”) consisting of:
  - The empty tuple: (),
  - Or a tuple containing two values, each of which is an integer or a numeric pair (still more recursion!)

• Given such a numeric pair, how many numbers are in it?
Example: Code

def count_vals(pair):
    """Assuming PAIR is a numeric pair, the total number of integers
    contained in the pair.
    >>> count_vals(())
    0
    >>> count_vals((1, ()
    1
    >>> count_vals((1, 2)
    2
    >>> count_vals(((1, 2), ((3, 4), ())))
    4
    """
    if ________:
        return 0
    elif type(pair) is int:
        return _
    else return __________
def count_vals(pair):
    """Assuming PAIR is a numeric pair, the total number of integers
    contained in the pair.
    >>> count_vals(())
    0
    >>> count_vals((1, ()))
    1
    >>> count_vals((1, 2))
    2
    >>> count_vals(((1, 2), ((3, 4), ())))
    4
    """
    if pair == ():
        return 0
    elif type(pair) is int:
        return _
    else return ____________________________
def count_vals(pair):
    """Assuming PAIR is a numeric pair, the total number of integers contained in the pair.
>>> count_vals(())
0
>>> count_vals((1, ()))
1
>>> count_vals((1, 2))
2
>>> count_vals(((1, 2), ((3, 4), ())))
4
"""
    if pair == ():
        return 0
    elif type(pair) is int:
        return 1
    else:
        return ________________________________
Example: Code

def count_vals(pair):
    """Assuming PAIR is a numeric pair, the total number of integers
    contained in the pair.
    >>> count_vals(())
    0
    >>> count_vals( (1, ()) )
    1
    >>> count_vals( (1, 2) )
    2
    >>> count_vals( ((1, 2), ((3, 4), ())) )
    4
    """
    if pair == ():
        return 0
    elif type(pair) is int:
        return 1
    else:
        return #ints in pair[0] + #ints in pair[1]
Example: Code

def count_vals(pair):
    """Assuming PAIR is a numeric pair, the total number of integers contained in the pair.
    >>> count_vals(())
    0
    >>> count_vals( (1, ()) )
    1
    >>> count_vals( (1, 2) )
    2
    >>> count_vals( ((1, 2), ((3, 4), ())) )
    4
    """

    if pair == ():
        return 0
    elif type(pair) is int:
        return 1
    else:
        return count_vals(pair[0]) + count_vals(pair[1])
The Recursive Leap of Faith

• To implement count_vals, we trusted its comment to be correct, even as we implemented it.

• This is the essence of recursive thinking.

• If we can show that
  - Our implementation is correct given that the comment is correct,
  - And if we can show that the process must terminate,

then the comment (the specification of the function) is correct.

• For recursive data structures, showing termination involves using a form of Noetherian induction.
Noetherian Induction

- A relation on values is well-founded if there are no infinite descending chains:
  - That is, if you start at some value and keep stepping to smaller values (according to the relation), then you must always get to a minimal value after finite steps.
  - E.g., natural or positive numbers under $<$.
  - Or numeric pairs under “is an element of.”
- Principle of Noetherian induction (named after Emmy Noether):
  - If $P(x)$ is statement about values $x$ from a well-founded set, and
  - If $P(x)$ is true whenever $P(y)$ is true for all $y < x$,
  - Then $P(x)$ is true for all $x$. 

(Source: http://en.wikipedia.org/wiki/Emmy_Noether)
Induction and Recursion

• Recursive programs are justified (and constructed) by inductive reasoning.

• Basic structure:

```python
def f(x):
    if There are no valid values≺x:
        # The ‘‘base case’’
        return A value that’s correct when x is minimal
    else:
        # Use ‘‘The inductive hypothesis’’
        return A solution constructed using f(y) where y≺x
```

• The meaning of≺depends on the application.

• In place of “return” might also use side-effect-producing code.