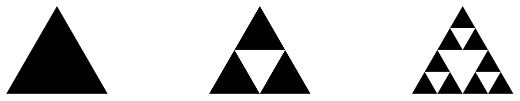
Lecture #18: Recursion

Subproblems and Self-Similarity

- Recursive routines arise when solving a problem naturally involves solving smaller instances of the same problem.
- With rlists, this happened because an rlist contains a reference to a smaller rlist: a recursive data structure.
- A classic example where the subproblems are visible is *Sierpinski's Triangle* (aka bit Sierpinski's Gasket).
- This triangle may be formed by repeatedly replacing a figure, initially a solid triangle, with three quarter-sized images of itself (1/2 size in each dimension), arranged in a triangle:



The Gasket in Python

- We can describe this as a recursive Python program (producing Postscript output).

```
sin60 = sqrt(3) / 2
def make_triangle(x, y, s, n):
    if n == 0:
        print("{0} {1} moveto {2} 0 rlineto "
              "-{3} {4} rlineto closepath fill"
              .format(x, y, s, s/2, s*sin60))
    else:
        make_triangle(x, y, s/2, n - 1)
        make_triangle(x + s/2, y, s/2, n - 1)
        make_triangle(x + s/4, y + sin60*s/2, s/2, n-1)

print("%!")
make_triangle(100, 100, 400, 8)
```

Aside: The Gasket in Postscript

- One can also perform the logic to generate figures in Postscript directly, which is itself a full-fledged programming language:

```
%!

/sin60 3 sqrt 2 div def

/make_triangle {
    dup 0 eq {
        3 index 3 index moveto 1 index 0 rlineto 0 2 index rlineto
                1 index neg 0 rlineto closepath fill
    } {
        3 index 3 index 3 index 0.5 mul 3 index 1 sub make_triangle
        3 index 2 index 0.5 mul add 3 index 3 index 0.5 mul
              3 index 1 sub make_triangle
        3 index 2 index 0.25 mul add 3 index 3 index 0.5 mul add
              3 index 0.5 mul 3 index 1 sub make_triangle
    } ifelse
    pop pop pop pop
} def

100 100 400 8 make_triangle showpage
```

Recursive Thinking

- When you call a function from the Python library, you don't look at its implementation, just its documentation ("the contract").
- By *recursive thinking,* I mean the extension of this same discipline to functions *as you are defining them.*
- Old example from Lecture #9: filtering an rlist:

```
def filter_rlist(cond, seq):
    """The subsequence of rlist 'seq' for which the 1-argument
    function 'cond' returns a true value."""
    if seq == empty_rlist:
        return empty_rlist
    elif cond(first(seq)):
        return make_rlist(first(seq),
                          Subseq. of rest(seq) satisfying cond)
    else:
        return Subseq. of rest(seq) satisfying cond
```

- How do we do the red parts? Look! This function filter_rlist says that it does what we need!

Recursive Thinking Used

- We take the "leap of faith," therefore, and use filter_rlist:

```
def filter_rlist(cond, seq):
    """The subsequence of rlist 'seq' for which the 1-argument
    function 'cond' returns a true value."""
    if seq == empty_rlist:
        return empty_rlist
    elif cond(first(seq)):
        return make_rlist(first(seq),
                          filter_rlist(cond, rest(seq)))
    else:
        return filter_rlist(cond, rest(seq))
```

Recursive Thinking in Mathematics

- To prevent an infinite recursion, must use this technique only when
  - The recursive cases are “smaller” than the input case, and
  - There is a minimum “size” to the data, and
  - All chains of progressively smaller cases reach a minimum in a finite number of steps.
- We say that such “smaller than” relations are *well founded*.
- We have

  Theorem (Noetherian Induction): Suppose $\prec$ is a well-founded relation and P is some property (predicate) such that whenever $P(y)$ is true for all $y \prec x$, then $P(x)$ is also true. Then $P(x)$ is true for all x.

  (After Emmy Noether 1882–1935, Göttingen and Bryn Mawr).

Example: Making Change

- Suppose that we have a sequence of coin or bill values (indicating what denominations of money exist) and an amount for which we wish to make change.
- First, how can we change the amount?

```
def make_change(amount, coins = (50, 25, 10, 5, 1)):
    """A sequence of integers giving a number of each type of coin
    in COINS such that the value of the indicated numbers of coins
    will by exactly AMOUNT.
    >>> tuple(make_change(81))
    (1, 1, 0, 1, 1)
    >>> tuple(make_change(47)
    (0, 1, 2, 0, 2)
    >>> tuple(make_change(47, (50, 25, 5, 1))
    (0, 1, 4, 2)
    """
```