Announcements

- Phase 1 of Hog project due Tuesday night (1 point).
- Submitting whole project by Thursday night earns 1 point of extra credit.
- Homework 2 due Thursday night,
- Lab party on Monday (2/1) from 4–5:30PM PT (see @430).
- Project party on Tuesday (2/2) from 7–8:30PM PT (see @430)
- Ask questions on the Piazza thread for today's lecture (@438).

Lecture #6: Recursion

Review: Philosophy of Functions (I)

Syntactic specification (signature)

```
def sqrt(x):
    """Assuming X >= 0,
    returns approximation to square root of X."""
```

Precondition

Postcondition

Semantic specification

- Specifies a *contract* between caller and function implementer.
- **Syntactic specification** gives syntax for calling (number of arguments).
- **Semantic specification** tells what it does:
  - **Preconditions** are requirements on the caller.
  - **Postconditions** are promises from the function's implementer.

Philosophy of Functions (II)

- Ideally, the specification (syntactic and semantic) should suffice to use the function (i.e., without seeing its body).
- There is a *separation of concerns* here:
  - The caller (client) is concerned with providing values of x and using the results, but *not* how the result is computed.
  - The implementer is concerned with how the result is computed, but *not* where x comes from or how the value is used.
  - From the client's point of view, `sqrt` is an *abstraction* from the set of possible ways to compute this result.
  - Therefore, we call this *functional abstraction.*
- Programming is largely about choosing abstractions that lead to clear, fast, and maintainable programs.

Simple Linear Recursions

- Take another look at the problem of adding squares of integers:

```
def sum_squares(N):
    """Return The sum of K**2 for K from 1 to N (inclusive)."""
    if ??:
        return 0            # When is this answer correct?
    else:
        return ??
```

Simple Linear Recursions

- Take another look at the problem of adding squares of integers:

```
def sum_squares(N):
    """Return The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return ??  # What if N >= 1?
```

Simple Linear Recursions

- Take another look at the problem of adding squares of integers:

```
def sum_squares(N):
    """Return The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return sum of K**2 for K from 1 to N-1 + N**2
```

Simple Linear Recursions

- Take another look at the problem of adding squares of integers:

```
def sum_squares(N):
    """Return The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else: # Use the comment!!
        return sum_squares(N - 1) + N**2
```

Simple Linear Recursions

- Take another look at the problem of adding squares of integers:

```
def sum_squares(N):
    """Return The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else: # Use the comment!!
        return sum_squares(N - 1) + N**2
```

- This is a simple *linear recursion*, with one recursive call per function instantiation.
- Can imagine a call as an expansion:

```
sum_squares(3) => sum_squares(2) + 3**2
               => sum_squares(1) + 2**2 + 3**2
               => sum_squares(0) + 1**2 + 2**2 + 3**2
               => 0 + 1**2 + 2**2 + 3**2 => 14
```

- Each call in this expansion corresponds to an environment frame, linked to the global frame, as shown in the Python Tutor.

Tail Recursion

- In Lecture #3, we saw a special kind of recursion that is strongly linked to iteration. Here is a variation of that example and a corresponding iterative version, with correspondences labeled (see boxes):

```
def sum_squares(N):
    """The sum of K**2
    for 1 <= K <= N."""
    accum = 0  A
    k = 1  B
    while k <= N:  C
        accum = accum + k**2  D
        k = k + 1  E
    return accum  F
```

```
def sum_squares(N):
    """The sum of K**2
    for 1 <= K <= N."""
    def part_sum(accum, k):
        if k <= N:  C
                                    D         E
            return part_sum(accum + k**2, k + 1)
        else:
            return accum  F
                     A    B
    return part_sum(0, 1)
```

- The right version is a *tail-recursive function*, meaning that the recursive call is either the returned value or the very last action performed.
- The values of the parameters on the right correspond to the values of the local variables on the left, as shown here.
- Essentially this same technique can be applied to **while** loops generally.

Recursive Thinking

- So far in this lecture, I've shown recursive functions by tracing or repeated expansion of their bodies.
- But when you call a function from the Python library, you don't look at its implementation, just its documentation ("the contract").
- *Recursive thinking* is the extension of this same discipline to functions *as you are defining them.*
- When implementing `sum_squares`, we reason as follows:
  - **Base case:** We know the answer is 0 if there is nothing to sum ($N < 1$).
  - Otherwise, we observe that the answer is N^2 plus the sum of the positive integers from 1 to $N - 1$.
  - But there is a function (`sum_squares`) that can compute $1 + \ldots + N - 1$ (its comment says so).
  - So when $N \geq 1$, we should return $N^2 + \texttt{sum\_squares}(N - 1)$.
- This "recursive leap of faith" works as long as we can guarantee we'll hit the base case.

Preventing Infinite Recursion

- To prevent an infinite recursion, we take the leap of faith 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 this minimum in a finite number of steps.
- We say that a set of values with such a "smaller than" relation is *well founded*.
- For example, the inputs can be a set of integers with a smallest member (like the non-negative or positive integers).
- Later, we'll see examples where the inputs are sequences of values and "smaller" means "shorter".

Induction

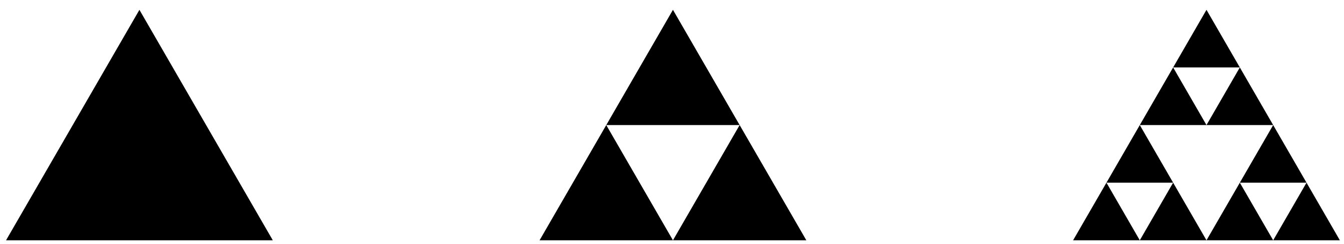
- In mathematics, we have various kinds of *induction* for proving that a property $P(x)$ holds for all x in some set.
- You have likely seen the familiar sort of line-of-dominoes induction on integers:
  - If $P(b)$ (the base case), and
  - $P(k-1)$ implies $P(k)$ for all $k > b$,
  - then $P(k)$ for all $k \geq b$.
- There is also a more general form called *well-founded* or *Noetherian induction*:
  - If P is some property (predicate) on a well-founded set with relation $\prec$ 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).
- The sets here can be anything with a partial ordering (such as lists ordered by length).

Recursion and Induction

- So recursive thinking is a kind of inductive thinking.
- The property we want to demonstrate is that our function works on inputs of any size.
- Induction tells us that we can conclude that our function works on all inputs if
  1. Whenever our function works on all arguments that are smaller than a given input, it must also work on that input,
  2. Any sequence of inputs in which each is smaller than its predecessor must eventually end.
  - The assumption that our function works on smaller arguments is the "recursive leap of faith."

Subproblems and Self-Similarity

- Recursive routines arise when solving a problem naturally involves solving smaller instances of the same problem.
- A classic example where the subproblems are visible is *Sierpinski's Triangle* (aka *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:



- Or we can describe creating a "triangle of order N and size S" as drawing either
  - a solid triangle with side S if $N = 0$, or
  - three triangles of size $S/2$ and order $N - 1$ arranged in a triangle.

The Gasket in Python

- We can write this description as a recursive Python program that produces Postscript output suitable for printing (see 06.py).

```
sin60 = sqrt(3) / 2
def make_gasket(n, x, y, s, output):
    """Write Postscript commands to OUTPUT that draw an Nth-order
    Sierpinski's gasket, with lower-left corner at (X,Y), and
    size S X S (units of points: 1/72 in)."""
    if n == 0:
        draw_solid_triangle(x, y, s, output)
    else:
        make_gasket(n - 1, x, y, s/2, output)
        make_gasket(n - 1, x + s/2, y, s/2, output)
        make_gasket(n - 1, x + s/4, y + sin60*s/2, s/2, output)

def draw_solid_triangle(x, y, s, output):
    """Draw a solid triangle lower-left corner at (X, Y)
    and side S on OUTPUT."""
    print(f"{x:.2f} {y:.2f} moveto "
          f"{s:.2f} 0 rlineto "
          f"-{s/2:.2f} {s*sin60:.2f} rlineto "
          "closepath fill", file=output)
```

Aside: The Gasket in Pure Postscript

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

```
%!

/sin60 3 sqrt 2 div def

/make_gasket {
    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_gasket
        3 index 2 index 0.5 mul add 3 index 3 index 0.5 mul
              3 index 1 sub make_gasket
        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_gasket
    } ifelse
    pop pop pop pop
} def

100 100 400 8 make_gasket showpage
```

Discussion

- The `make_gasket` function in Python is a more complex (and frankly, more representative) use of recursion than previous examples in this lecture.
- These previous examples were linear recursions, where each call of the function resulted in it making one recursive call before returning.
- The gasket example, however, makes three calls.
- Thus, it is an example of a *tree recursion*, our topic for next time.