

Lecture #9: More Functions

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 1

Another Tree Recursion: Hog Dice

- What are the odds of rolling at least k in hog with n s -sided dice? ($n > 0$ and for us, $s > 0$ is 4 or 6)

$$\frac{\text{\# rolls of } n \text{ } s\text{-sided dice totaling } \geq k}{s^n}$$

- If $k \leq 1$, then clearly the numerator is just s^n .
- For $k > 1$, we consider only rolls that include dice values $2-s$, since any 1-die "pigs out." Let's call this quantity `rolls2(k, n, s)`.
- The number of ways to score $\geq k$ is 0 if $ns < k$. This is a base case.
- If $n > 0$ then the number of ways to score at least $k \leq 1$ with n dice none of which is 1 is $(s-1)^n$. This is also a base case.
- If the first die comes up d ($2 \leq d \leq s$), then there are `rolls2(k-d, n-1, s)` ways to throw the remaining $n-1$ dice to get a total of at least k with all n dice.
- This gives us a tree recursion. How would you modify it for the "swine swap" rule?

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 2

Back to Numeric Pairs: Find the Number

- A **numeric pair** is either an empty tuple, an integer, or a tuple consisting of two numeric pairs (slight revision from last time).
- Problem: does the number x occur in a given numeric pair?

```
def occurs(x, pair):
    """X occurs at least once in numeric pair PAIR.
    >>> occurs(3, ((2, 1), ((), (3, ())))))
    True
    >>> occurs(5, ((2, 1), ((), (3, ())))))
    False
    """
    if x == pair:
        return True
    elif pair == () or type(pair) is int:
        return False
    else:
        return occurs(x, pair[0]) or occurs(x, pair[1])
```

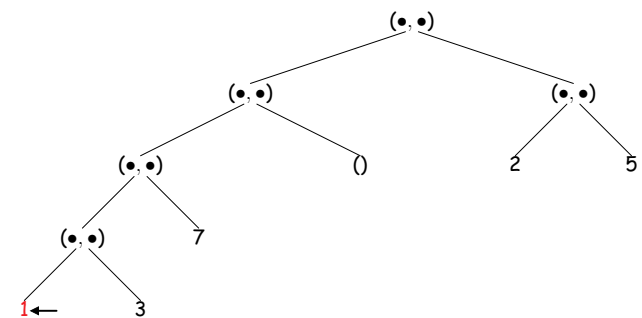
- What is the time required by this function proportional to? A: The total number of tuples and integers in pair.

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 3

Numeric Pairs: First Leaf

- A **leaf** in a numeric pair is the empty tuple or an integer.
- Define the **first leaf** as the leftmost leaf in the Python expression that denotes a tree.
- Example: the first leaf of `((((1, 3), 7), ()), (2, 5))` is 1:



Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 4

First Leaf Code

```
def first_leaf(pair):
    """The first leaf in PAIR, reading left to right.
    >>> first_leaf(())
    ()
    >>> first_leaf(5)
    5
    >>> first_leaf(((3, ()), (2, 1)), (()))
    3
    >>> first_leaf(((((), 3), (2, 1)), ()))
    ()
    """
    if type(pair) is int or pair == ():
        return pair
    else:
        return first_leaf(pair[0])
```

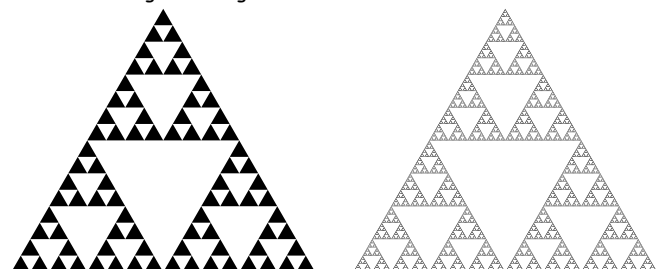
What kind of a recursive process is this? A: Iterative process (tail recursion)

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 5

Sierpinski Triangle

- No discussion of recursion is complete without a mention of **fractal patterns**, which exhibit self-similarity when scaled.
- We'll define a "Sierpinski Triangle of depth k and side s " to be
 - A filled equilateral triangle with sides of length s , if $k = 0$, else
 - Three Sierpinski Triangles of depth $k-1$ and side $s/2$ arranged in the three corners of an equilateral triangle with side s .
- Here are triangles of degree 4 and 8:



Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 6

Drawing Sierpinski Triangles

- Assume the existence of the function `triangle`:

```
def triangle(x, y, side):
    """Draw a filled equilateral triangle with its lower-left corner
    at (X, Y) and with given SIDE. The base is aligned with the x-axis."""
```

- We can now read off the definition of the triangle:

```
def sierpinski(x, y, side, depth):
    """Draw a Sierpinski triangle of given DEPTH with given SIDE and
    lower-left corner at (X, Y)."""

    if depth == 0:
        triangle(x, y, side)
    else:
        height = 0.25 * sqrt(3) * side

        sierpinski(x, y, side/2, depth-1)
        sierpinski(x + side/4, y + height, side/2, depth-1)
        sierpinski(x + side/2, y, side/2, depth-1)
```

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 7

Functions: Separation of Concerns

- The `sierpinski` routine used `triangle`.
- To write `sierpinski`, I needed only to know:
 - The *syntactic specification* of `triangle`: its name and number of arguments (given by its `def` header), and
 - Its *semantic specification*: what a call does or means (given by its documentation comment).
- I did **not** need to know how `triangle` works or who else calls it.
- Likewise, `triangle` does **not** need to know
 - where its arguments come from,
 - who calls it, or
 - what use is made of its return value or side effects.
- There is a *separation of concerns* between these functions.
- This is a fundamental concept in software engineering: organize programs so that you can work on one thing at a time in isolation.

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 8

Names

Semantically, names are arbitrary: to the reader, they are part of the documentation.

Bad:	Better:
number	dice_rolls
true_false	pigged_out
d	dice, die
helper	take_turn, find_repeat
do_stuff	rescale_figure
random obscenity	report_error
I, I, O	k, m, n

Names convey meaning or purpose to the programmer (not to the machine).

Function names should convey their value (`abs`, `sqrt`) or effect (`print`)

Use the documentation comments of functions to elaborate where necessary, to indicate the types of arguments and return values, and to indicate assumptions or limitations on the arguments.

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 9

Names and Comments

- I generally limit comments to
 - Docstrings on functions (or later, on classes)
 - Comments and documentation at the beginning of a module describing its purpose, conventions, authorship, copyright permissions, etc.
 - Comment names of significant constants.
- Avoid internal comments: they indicate places where you could make a function shorter or use a better name:

Rather than

Use

```
# Compute the discriminant
d = b**2 - 4*a*c
```

```
discriminant = b**2 - 4*a*c
```

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 10

Function Comments

Comments on a function should suffice to tell the reader everything needed to use it.

Rather than

Use

```
def largest(L):
    """Find the largest value"""
    k = 0
    for i in range(1, len(L)):
        if L[i] > L[k]:
            k = i
    return k
```

```
def largest(L):
    """Return the index of the largest
    value in L."""
    k = 0
    for i in range(1, len(L)):
        if L[i] > L[k]:
            k = i
    return k
```

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 11

Refactoring

- Your comments can suggest to you that things are getting too big, or that a function is doing too much.
- When that happens, it is time to *refactor*: break functions up into more coherent pieces.
- Consider the function:

```
def print_averages(grade_book, out):
    """Compute the average scores for each student in
    GRADE_BOOK and prints on OUT."""
```
- What if we just want to know the averages?
- What if we also want a different format, including other information?
- Makes more sense, e.g., to have a `get_averages` function, and a more general print routine that will print any information about students.

Last modified: Mon Feb 10 13:59:06 2014

CS61A: Lecture #9 12

Unit Testing

- The docstring tests that you execute with `python3 -m doctest` are examples of *unit tests*.
- That is, tests on the smallest testable units of your program (functions).
- *Test-driven development* refers to the practice of creating tests *ahead of* implementation.
- You don't have to wait for your program to be implemented to test it.
- The doctest Python module makes it possible to run all your tests cumulatively, watching for inadvertant errors and tracking how much still needs to be done.

Decorators

- You've seen functions on functions. They can also be used for testing or debugging:

```
def trace1(fn):
    """Return a function equivalent to FN, a one-argument
    function, that also prints trace output.
    """
    def traced(x):
        print('Calling', fn, 'on argument', x)
        return fn(x)
    return traced
```

- To use this:

```
def triple(x):
    return 3*x
triple = trace1(triple)
```

- Or, more conveniently, use Python's decorators:

```
@trace1
def triple(x):
    return 3*x
```