Another Recursion Problem: Counting Partitions

I'd like to know the number of distinct ways of expressing an integer as a sum of positive integer "parts."

To make things more interesting, let's also limit the size of the integer parts to some given value:

```python
def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)"""
    if n:
        return 0
    elif n == 1:
        return 1
    else:
        return:
```

Identifying the Problem in the Problem

Again, consider `num_partitions(6, 3)`.

Some partitions will contain the maximum size integer, 3, and the rest won't.

Those that contain 3 then have various ways to partition the remaining 3.

3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1

While those that do not contain 3 partition 6 using integers no larger than 2:

2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1

These observations generalize, and lead immediately to a solution.

for p in range(2, n):
    print(p)
### Functions and Data

#### Functions and Data (I)

- **Data Abstraction Philosophy**
  - For each type, we define an interface that describes for users what operations are available.
  - We call ADTs abstract data type (ADT).
  - We tend to think of functions as simply doing or computing some thing with data.
  - Doing abstraction is about defining a new data type.

- **The Pair Abstraction**
  - Together, these three functions define a data type.
  - A pair is a kind of thing and the operations upon it.
  - In the old days, one described programs as hierarchies of actions: procedural decomposition.

- **Rational Numbers**
  - Here's a small example of a pair:
  ```python
  def left(x):
      return x
  def right(x):
      return x
  def pair(left, right):
      return (right, left)
  ```

- **Counting Partitions: Code (IV)**

- **Data Abstraction Philosophy**

- **Trivial example:**
  ```python
  def left(x):
      return x
  def right(x):
      return x
  def pair(x):
      return (x, x)
  ```

#### Functions and Data (II)

- **Utility functions:**

  ```python
  def add(x, y):
      return x + y
  def mul(x, y):
      return x * y
  def const(x, y):
      return x
  ```

- **Trivial example:**

  ```python
  def add(x, y):
      return x + y
  def cons(x, y):
      return (x, y)
  ```

- **Code sample:**

  ```python
  def x():
      return 1
  def y():
      return 2
  ```

- **Trivial example:**

  ```python
  def add(x, y):
      return x + y
  ```
A Better Specification

• Problem is that "the numerator (denominator) of $r$" is not well-defined for a rational number.

• If `make_rat` really produced rational numbers, then `make_rat(2, 4)` and `make_rat(1, 2)` ought to be identical. So should `make_rat(1, -1)` and `make_rat(-1, 1)`.

• So a better specification would be

```python
def numer(r):
    """The numerator of rational number r in lowest terms."""

def denom(r):
    """The denominator of rational number r in lowest terms. Always positive."""
```

Rationals as Pairs (I)

• Our pair abstraction (represented by functions) can in turn represent rational numbers.

```python
from math import gcd
# Need Python3.5 actually.

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d)
    if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return cons(n, d)

def numer(r):
    """The numerator of rational number r."""
    return left(r)

def denom(r):
    """The denominator of rational number r."""
    return right(r)
```

Representation as Functions (II)

• One possibility for `add_rat`:

```python
def add_rat(x, y):
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                   denom(x) * denom(y))

def mul_rat(x, y):
    """The product of rational numbers x and y."""
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Changing Representations

• It's cute that functions can represent pairs (or anything else, for that matter), but it's not a particularly efficient use of them.

• Suppose that we instead decide to use Python's tuples. What changes?

```python
def cons(left, right):
    return (left, right)

def left(pair):
    return pair[0]

def right(pair):
    return pair[1]
```

• Crucial Observation: Nothing else changes!