Functions as Arguments

Consider the following three functions, which all compute summations. The first, `sum_naturals`, computes the sum of natural numbers up to `n`:

```
>>> def sum_naturals(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total
>>> sum_naturals(100)
5050
```

The second, `sum_cubes`, computes the sum of the cubes of natural numbers up to `n`.

```
>>> def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + pow(k, 3), k + 1
        return total
>>> sum_cubes(100)
25502500
```

The third, `pi_sum`, computes the sum of terms in the series

$$\frac{8}{1 \cdot 3} + \frac{8}{5 \cdot 7} + \frac{8}{9 \cdot 11} + \ldots$$

which converges to pi very slowly.

```
>>> def pi_sum(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + 8 / (k * (k + 2)), k + 4
        return total
>>> pi_sum(100)
3.121594652591009
```

These three functions clearly share a common underlying pattern. They are for the most part identical, differing only in name, the function of `k` used to compute the term to be added, and the function that provides the next value of `k`. We could generate each of the functions by filling in slots in the same template:

```
def <name>(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + <term>(k), <next>(k)
    return total
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Each of these functions is a summation of terms. As program designers, we would like our language to be powerful enough so that we can write a function that expresses the concept of summation itself rather than only functions that compute particular sums. We can do so readily in Python by taking the common template shown above and transforming the "slots" into formal parameters:

In the example below, `summation` takes as its three arguments the upper bound `n` together with the functions `term` and `next`. We can use `summation` just as we would any function, and it expresses summations succinctly. Take the time to step through this example, and notice how binding `cube` and `successor` to the local names `term` and `next` ensures that the result `1*1*1 + 2*2*2 + 3*3*3 = 36` is computed correctly. In this example, frames which are no longer needed are removed to save space.

```
1  def summation(n, term, next):
2      total, k = 0, 1
3      while k <= n:
4          total, k = total + term(k), next(k)
5      return total
6
```

```
7  def cube(k):
8      return pow(k, 3)
9
10 def successor(k):
11     return k + 1
12
13 def sum_cubes(n):
14     return summation(n, cube, successor)
15
16 result = sum_cubes(3)
```

Using an `identity` function that returns its argument, we can also sum natural numbers.

```
>>> def identity(k):
        return k
>>> def sum_naturals(n):
        return summation(n, identity, successor)
>>> sum_naturals(10)
55
```

We can define `pi_sum` in terms of `term` and `next` functions, using our `summation` abstraction to combine components. We pass the argument `1e6`, a shorthand for `1 * 10^6 = 1000000`, to generate a close approximation to pi.

```
>>> def pi_term(k):
        denominator = k * (k + 2)
        return 8 / denominator
>>> def pi_next(k):
        return k + 4
>>> def pi_sum(n):
```

```
        return summation(n, pi_term, pi_next)
>>> pi_sum(1e6)
3.1415906535898936
```

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#functions-as-arguments