Functions as Returned Values and Currying in Python

Functions as Returned Values :

We can achieve even more expressive power in our programs by creating functions whose returned values are themselves functions. An important feature of lexically scoped programming languages is that locally defined functions maintain their parent environment when they are returned. The following example illustrates the utility of this feature.

Once many simple functions are defined, function *composition* is a natural method of combination to include in our programming language. That is, given two functions `f(x)` and `g(x)`, we might want to define `h(x) = f(g(x))`. We can define function composition using our existing tools:

```
>>> def compose1(f, g):
        def h(x):
            return f(g(x))
        return h
```

The environment diagram for this example shows how the names `f` and `g` are resolved correctly, even in the presence of conflicting names.

```
def square(x):
    return x * x

def successor(x):
    return x + 1
```

```

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

def f(x):
    """A function named f that is never called."""
    return -x

add_one_and_square = compose1(square, successor)
result = add_one_and_square(12)
```

The `1` in `compose1` is meant to signify that the composed functions all take a single argument. This naming convention is not enforced by the interpreter; the `1` is just part of the function name.

At this point, we begin to observe the benefits of our effort to define precisely the environment model of computation. No modification to our environment model is required to explain our ability to return functions in this way.

Currying :

We can use higher-order functions to convert a function that takes multiple arguments into a chain of functions that each take a single argument. More specifically, given a function `f(x, y)`, we can define a function `g` such that `g(x)(y)` is equivalent to`f(x, y)`.

Here, `g` is a higher-order function that takes in a single argument `x` and returns another function that takes in a single argument `y`. This transformation is called *currying*.

As an example, we can define a curried version of the `pow` function:

```
>>> def curried_pow(x):
        def h(y):
            return pow(x, y)
        return h
>>> curried_pow(2)(3)
8
```

Some programming languages, such as Haskell, only allow functions that take a single argument, so the programmer must curry all multi-argument procedures. In more general languages such as Python, currying is useful when we require a function that takes in only a single argument. For example, the *map* pattern applies a single-argument function to a sequence of values. In later chapters, we will see more general examples of the map pattern, but for now, we can implement the pattern in a function:

```
>>> def map_to_range(start, end, f):
        while start < end:
            print(f(start))
            start = start + 1
```

We can use `map_to_range` and `curried_pow` to compute the first ten powers of two, rather than specifically writing a function to do so:

```
>>> map_to_range(0, 10, curried_pow(2))
1
2
4
8
16
32
64
128
256
512
```

We can similarly use the same two functions to compute powers of other numbers. Currying allows us to do so without writing a specific function for each number whose powers we wish to compute.

In the above examples, we manually performed the currying transformation on the `pow` function to obtain `curried_pow`. Instead, we can define functions to automate currying, as well as the inverse *uncurrying* transformation:

```
>>> def curry2(f):
        """Return a curried version of the given two-
argument function."""
        def g(x):
            def h(y):
                return f(x, y)
            return h
        return g
>>> def uncurry2(g):
        """Return a two-argument version of the given
curried function."""
        def f(x, y):
            return g(x)(y)
        return f
>>> a = curry2(pow)
>>> map_to_range(0, 10, a(2))
1
2
4
8
16
32
64
128
256
512
```

The `curry2` function takes in a two-argument function `f` and returns a single-argument function `g`. When `g` is applied to an argument `x`, it returns a single-argument function `h`. When `h` is applied to `y`, it calls `f(x, y)`. Thus, `curry2(f)(x)(y)` is equivalent to `f(x, y)`, so `curry2(f)` returns a curried version of `f`.

The `uncurry2` function should reverse the currying transformation, so that `uncurry2(curry2(f))` is equivalent to `f`. We leave the argument that this is indeed the case as an exercise.

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