Functions as General Methods in Python

We introduced user-defined functions as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order functions, we begin to see a more powerful kind of abstraction: some functions express general methods of computation, independent of the particular functions they call.

Despite this conceptual extension of what a function means, our environment model of how to evaluate a call expression extends gracefully to the case of higher-order functions, without change. When a user-defined function is applied to some arguments, the formal parameters are bound to the values of those arguments (which may be functions) in a new local frame.

Consider the following example, which implements a general method for iterative improvement and uses it to compute the golden ratio. An iterative improvement algorithm begins with a `guess` of a solution to an equation. It repeatedly applies an `update`function to improve that guess, and applies an `isclose` comparison to check whether the current `guess` is "close enough" to be considered correct.

```
>>> def improve(update, isclose, guess=1):
        while not isclose(guess):
            guess = update(guess)
        return guess
```

One way to know if the current guess "`isclose`" is to check whether two functions, `f` and `g`, are near to each other for that guess. Testing whether `f(x)` is `near` to `g(x)` is again a general method of computation.

```
>>> def near(x, f, g):
        return approx_eq(f(x), g(x))
```

A common way to test for approximate equality in programs is to compare the absolute value of the difference between numbers to a small tolerance value.

```
>>> def approx_eq(x, y, tolerance=1e-3):
        return abs(x - y) < tolerance
```

The golden ratio, often called "phi", is a number that appears frequently in nature, art, and architecture. It can be computed via`improve` using the `golden_update`, and it converges when its successor is equal to its square.

```
>>> def golden_update(guess):
        return 1/guess + 1
>>> def square_near_successor(guess):
        return near(guess, square, successor)
```

Calling `improve` with the arguments `golden_update` and `square_near_successor` will compute an approximation to the golden ratio.

```
>>> improve(golden_update, square_near_successor)
1.6180371352785146
```

By tracing through the steps of evaluation, we can see how this result is computed. First, a local frame for `improve` is constructed with bindings for `update`, `isclose`, and `guess`. In the body of `improve`, the name `isclose` is bound to`square_near_successor`, which is called on the initial value of `guess`. In turn, `square_near_successor` calls `near`, creating a third local frame that binds the formal parameters `f` and `g` to `square` and `successor`.

```
1  def square(x):
2      return x * x
3
4  def successor(x):
5      return x + 1
6
7  def improve(update, isclose, guess=1):
```

```
    while not isclose(guess):
        guess = update(guess)
    return guess

def near(x, f, g):
    return approx_eq(f(x), g(x))

def approx_eq(x, y, tolerance=1e-3):
    return abs(x - y) < tolerance

def golden_update(guess):
    return 1/guess + 1

def square_near_successor(guess):
    return near(guess, square, successor)

phi = improve(golden_update, square_near_successor)
```


Completing the evaluation of `near`, we see that the `square_near_successor` is `False` because 1 is not close to 2. Hence, evaluation proceeds with the suite of the `while` clause, and this mechanical process repeats several times.

This extended example illustrates two related big ideas in computer science. First, naming and functions allow us to abstract away a vast amount of complexity. While each function definition has been trivial, the computational process set in motion by our evaluation procedure appears quite intricate. Second, it is only by virtue of the fact that we have an extremely general evaluation procedure that small components can be composed into complex processes. Understanding that procedure allows us to validate and inspect the process we have created.

As always, our new general method `improve` needs a test to check its correctness. The golden ratio can provide such a test, because it also has an exact closed-form solution, which we can compare to this iterative result.

```
>>> phi = 1/2 + pow(5, 1/2)/2
>>> def near_test():
        assert near(phi, square, successor), 'phi * phi
is not near phi + 1'
>>> def improve_test():
        approx_phi = improve(golden_update,
square_near_successor)
        assert approx_eq(phi, approx_phi), 'phi differs
from its approximation'
```

Extra for experts. We left out a step in the justification of our test. For what range of tolerance values `e` can you prove that if `near(x, square, successor)` is true with `tolerance` value `e`, then `approx_eq(phi, x)` is true with the same tolerance?

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