Abstractions, First-Class Functions and Function Decorators in Python

Abstractions and First-Class Functions :

We began this section with the observation that user-defined functions are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order functions permit us to manipulate these general methods to create further abstractions.

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs, build upon them, and generalize them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order functions is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the "rights and privileges" of first-class elements are:

1. They may be bound to names.
2. They may be passed as arguments to functions.
3. They may be returned as the results of functions.
4. They may be included in data structures.

Python awards functions full first-class status, and the resulting gain in expressive power is enormous.

Function Decorators :

Python provides special syntax to apply higher-order functions as part of executing a `def` statement, called a decorator. Perhaps the most common example is a trace.

```
>>> def trace1(fn):
        def wrapped(x):
            print('-> ', fn, '(', x, ')')
            return fn(x)
        return wrapped
>>> @trace1
    def triple(x):
        return 3 * x
>>> triple(12)
->  <function triple at 0x102a39848> ( 12 )
36
```

In this example, A higher-order function `trace1` is defined, which returns a function that precedes a call to its argument with a`print` statement that outputs the argument. The `def` statement for `triple` has an annototation, `@trace1`, which affects the execution rule for `def`. As usual, the function `triple` is created. However, the name `triple` is not bound to this function. Instead, the name `triple` is bound to the returned function value of calling `trace1` on the newly defined `triple` function. In code, this decorator is equivalent to:

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

In the projects associated with this text, decorators are used for tracing, as well as selecting which functions to call when a program is run from the command line.

Extra for experts. The decorator symbol `@` may also be followed by a call expression. The expression following `@` is evaluated first (just as the name `trace` was evaluated

above), the `def` statement second, and finally the result of evaluating the decorator expression is applied to the newly defined function, and the result is bound to the name in the `def` statement

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#abstractions-and-first-class-functions