Functions in Python - Part II

Local Names

One detail of a function's implementation that should not affect the function's behavior is the implementer's choice of names for the function's formal parameters. Thus, the following functions should provide the same behavior:

```
>>> def square(x):
        return mul(x, x)
>>> def square(y):
        return mul(y, y)
```

This principle -- that the meaning of a function should be independent of the parameter names chosen by its author -- has important consequences for programming languages. The simplest consequence is that the parameter names of a function must remain local to the body of the function.

If the parameters were not local to the bodies of their respective functions, then the parameter `x` in `square` could be confused with the parameter `x` in `sum_squares`. Critically, this is not the case: the binding for `x` in different local frames are unrelated. The model of computation is carefully designed to ensure this independence.

We say that the *scope* of a local name is limited to the body of the user-defined function that defines it. When a name is no longer accessible, it is out of scope. This scoping behavior isn't a new fact about our model; it is a consequence of the way environments work.

Practical Guidance: Choosing Names

The interchangeabily of names does not imply that formal parameter names do not matter at all. On the contrary, well-chosen function and parameter names are essential for the human interpretability of function definitions!

The following guidelines are adapted from the style guide for Python code, which serves as a guide for all (non-rebellious) Python programmers. A shared set of conventions smooths communication among members of a developer community. As a side effect of following these conventions, you will find that your code becomes more internally consistent.

1. Function names are lowercase, with words separated by underscores. Descriptive names are encouraged.
2. Function names typically evoke operations applied to arguments by the interpreter (e.g., `print`, `add`, `square`) or the name of the quantity that results (e.g., `max`, `abs`, `sum`).
3. Parameter names are lowercase, with words separated by underscores. Single-word names are preferred.
4. Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed.
5. Single letter parameter names are acceptable when their role is obvious, but avoid "l" (lowercase ell), "O" (capital oh), or "I" (capital i) to avoid confusion with numerals.

There are many exceptions to these guidelines, even in the Python standard library. Like the vocabulary of the English language, Python has inherited words from a variety of contributors, and the result is not always consistent.

Functions as Abstractions

Though it is very simple, `sum_squares` exemplifies the most powerful property of user-defined functions. The function`sum_squares` is defined in terms of the function `square`, but relies only on the relationship that `square` defines between its input arguments and its output values.

We can write `sum_squares` without concerning ourselves with *how* to square a number. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as `sum_squares` is concerned, `square` is not a particular function body, but rather an abstraction of a function, a so-called functional abstraction. At this level of abstraction, any function that computes the square is equally good.

Thus, considering only the values they return, the following two functions for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.

```
>>> def square(x):
        return mul(x, x)
>>> def square(x):
        return mul(x, x-1) + x
```

In other words, a function definition should be able to suppress details. The users of the function may not have written the function themselves, but may have obtained it from another programmer as a "black box". A programmer should not need to know how the function is implemented in order to use it. The Python Library has this property. Many developers use the functions defined there, but few ever inspect their implementation.

To master the use of a functional abstraction, it is often useful to consider its three core attributes. The *domain* of a function is the set of arguments it can take. The *range* of a function is the set of values it can return. The *intent* of a function is the relationship it computes between inputs and output (as well as any side effects it might generate). Understanding functions via their domain, range, and intent is critical to using them correctly in a complex program.

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