

```
>>> from math import sqrt
>>> exec "sqrt = 1"
>>> sqrt(4)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in ?
    sqrt(4)
TypeError: object is not callable: 1
```

Well, why would you do something like that in the first place, you ask? The `exec` statement is mainly useful when you build the code string on the fly. And if the string is built from parts that you get from other places, and possibly from the user, you can rarely be certain of exactly what it will contain. So to be safe, you give it a dictionary, which will work as a namespace for it.

Note The concept of namespaces, or scopes, is a very important one. You will look at it in depth in the next chapter, but for now you can think of a namespace as a place where you keep your variables, much like an invisible dictionary. So when you execute an assignment like `x = 1`, you store the key `x` with the value `1` in the current namespace, which will often be the global namespace (which we have been using, for the most part, up until now), but doesn't have to be.

You do this by adding `in <scope>`, where `<scope>` is some dictionary that will function as the namespace for your code string:

```
>>> from math import sqrt
>>> scope = {}
>>> exec 'sqrt = 1' in scope
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

As you can see, the potentially destructive code does not overwrite the `sqrt` function; the function works just like it should, and the `sqrt` variable resulting from the `exec`'ed assignment is available from the scope.

Note that if you try to print out `scope`, you see that it contains a *lot* of stuff because the dictionary called `__builtins__` is automatically added and contains all built-in functions and values:

```
>>> len(scope)
2
>>> scope.keys()
['sqrt', '__builtins__']
```

`eval`

A built-in function that is similar to `exec` is `eval` (for “evaluate”). Just as `exec` executes a series of Python *statements*, `eval` evaluates a Python *expression* (written in a string) and returns the