Calling User-Defined Functions in Python

To evaluate a call expression whose operator names a user-defined function, the Python interpreter follows a computational process. As with any call expression, the interpreter evaluates the operator and operand expressions, and then applies the named function to the resulting arguments.

Applying a user-defined function introduces a second *local* frame, which is only accessible to that function. To apply a user-defined function to some arguments:

1. Bind the arguments to the names of the function's formal parameters in a new *local* frame.
2. Execute the body of the function in the environment that starts with this frame.

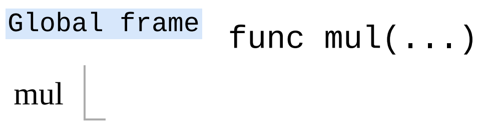
The environment in which the body is evaluated consists of two frames: first the local frame that contains formal parameter bindings, then the global frame that contains everything else. Each instance of a function application has its own independent local frame.

To illustrate an example in detail, several steps of the environment diagram for the same example are depicted below. After executing the first import statement, only the name `mul` is bound in the global frame.

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

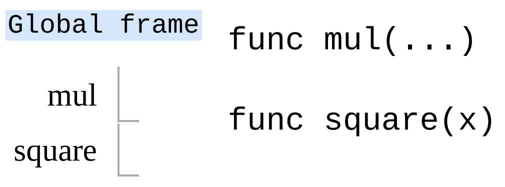
Edit code

< Back Step 2 of 5 Forward >



First, the definition statement for the function `square` is executed. Notice that the entire `def` statement is processed in a single step. The body of a function is not executed until the function is called (not when it is defined).

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

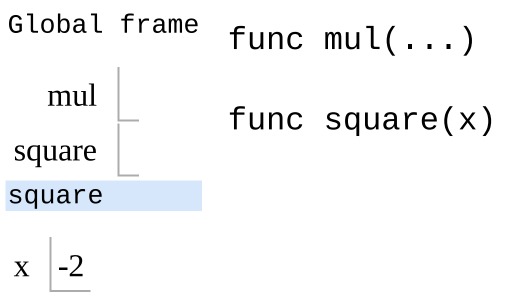


Edit code

< Back Step 3 of 5 Forward >

Next, The `square` function is called with the argument `-2`, and so a new frame is created with the formal parameter `x` bound to the value `-2`.

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

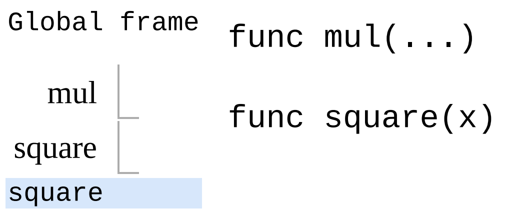


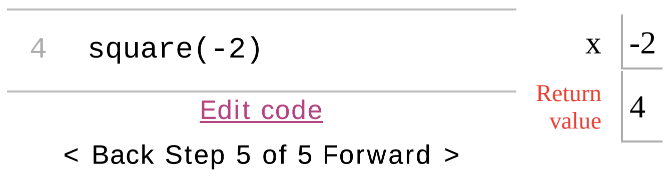
Edit code

< Back Step 4 of 5 Forward >

Then, the name `x` is looked up in the current environment, which consists of the two frames shown. In both occurrences, `x` evaluates to `-2`, and so the `square` function returns `4`.

```
from operator import mul
def square(x):
    return mul(x, x)
```





```
square(-2)
```

The "Return value" in the `square()` frame is not a name binding; instead it indicates the value returned by the function call that created the frame.

Even in this simple example, two different environments are used. The top-level expression `square(-2)` is evaluated in the global environment, while the return expression `mul(x, x)` is evaluated in the environment created for by calling `square`. Both `x` and`mul` are bound in this environment, but in different frames.

The order of frames in an environment affects the value returned by looking up a name in an expression. We stated previously that a name is evaluated to the value associated with that name in the current environment. We can now be more precise:

- A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Our conceptual framework of environments, names, and functions constitutes a *model of evaluation*; while some mechanical details are still unspecified (e.g., how a binding is implemented), our model does precisely and correctly describe how the interpreter evaluates call expressions. In Chapter 3 we will see how this model can serve as a blueprint for implementing a working interpreter for a programming language.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#calling-user-defined-functions