Wiki - Level 2 - Functions

Example: “double”

In several of the previous examples such as “fahrenheit_to_celsius”, we constructed a chunk of Python code that performed some useful computation that we might wish to carry out multiple times. Instead of copying the code, the standard solution in modern computing is to define a function that carries out the action of the code. For the “fahrenheit_to_celsius” example, this function would take fahrenheit as input and then compute and return the corresponding Celsius temperature. Here is a simple example of a function double defined via a function definition statement of the form

```
def double(x):
    return 2 * x
```

The indented statement in this example is the body of the function. (In general, the body of a Python function is a sequence of indented statements.) The return statement instructs Python that 2 * x is the value returned by the function double. Given this definition, we can call the function double later in the program via an expression of the form double(2). The example "double" illustrates this process.

```
# A function that doubles a number
def double(x):
    return 2 * x
print double(1 + 1)
```

The process by which Python evaluates a function call is fairly straightforward, but has multiple steps. We will use *Pystep* to illustrate this process visually. In particular, let's step through the program above. The first step of execution defines the function double. Note that the variable pane now has a definition for double of type <funct>. The next step evaluates the right-hand side of the print statement, double(1 + 1). To evaluate the expression double(1 + 1), Python first evaluates its argument 1 + 1 and reduces the expression to double(2).

Next, Python evaluates the function call double(2). This evaluation is by far the most complex process that we have studied up to now. To make this process more transparent, we will explain this process as a transformation into an equivalent Python program involving the body of the function and some extra assignment statements. It is important to note that, in reality, Python does not evaluate a function in this manner. Instead, the following explanation is designed to provide a simple equivalent mental model for the novice seeking to understand function evaluation.

To visualize the first step of evaluation of the function call double(2), *Pystep* applies the following transformations to the program:

- Replace the function call double(2) by a new variable double_0,
- Prepend the body of double to the top of the current program,
- Create an assignment of the parameter x to the corresponding value 2 from the function call,
- Prepend this assignment to the top of the current program,
- Replace the return 2 * x statement by a pair of statements, double_0 = 2 * x and return double_0.

The resulting program has the following form:

x_1 = 2
$double_0$ = 2 * x_1
return $double_0$
print $double_0$

You should use “Step” and “Unstep” to move back and forth between the program before the body of double is inserted and after the body of is inserted. Walk through the items above to make sure that you understand the transformation entailed in this single step.

Before proceeding with the evaluation of this program, we note that the subscripted variables introduced into the program are new variables created to store values generated and used in evaluating double. The subscript indicates the call level associated with the variable (the number of function calls pending when the variable was created) . Observe that the variable x_1 is local to the definition of double since x_1 is used purely inside the definition of double. $double_0$ is a temporary variable that stores the value returned by function evaluation for later use at the function call location. This temporary variable has subscript zero to indicate that it exists outside the body of the function call. (Note that $double_0$ is an artifact of our visualization of function evaluation. In actual Python function evaluation, the return value is directly substituted for the function call.)

At this point, evaluation of the program via stepping proceeds as usual. Observe that *Pystep* has inserted a blue line to separate the code add to the front of the program in response to the function call from the remainder of the code. When *Pystep* executes a return statement, this statement alerts Python that evaluation of the function is completed and Python then deletes all code up to the blue line. At this point, the variable $double_0$ contains the value returned by the function call and the print statement then prints 4 in the console.

Examples: “fahrenheit_to_kelvin”, “triangle_area_function”

Functions in Python may call other functions leading to a sequence of pending function calls as illustrated during the evaluation of the “fahrenheit_to_kelvin” example. In that example, we have defined two functions: one function f2c that converts temperatures from degrees Fahrenheit to degrees Celsius and a second function f2k that converts from degrees Fahrenheit to Kelvin with the help of f2c. Note that functions may also have more than one parameter such as illustrated in the “triangle_area_function” example.

Example: “print_name”

Besides returning a value, Python functions may also have side effects like printing out a message. In the example, “print_name”, we provide a function that take the string name and prints the string expression "My name is " + name to the console. In this example, the call to the function print_name is both an expression and a statement. Since function calls need not return a value, we use the return statement without a return expression to return from the function call. In this case, Python returns the special value None by default to indicate that no value was returned.