2 Procedures

In Python, the fundamental abstraction of a computation is as a procedure (other books call them "functions" instead; we'll end up using both terms). A procedure that takes a number as an argument and returns the argument value plus 1 is defined as:

```
def f(x):
    return x + 1
```

The indentation is important here, too. All of the statements of the procedure have to be indented one level below the `def`. It is crucial to remember the `return` statement at the end, if you want your procedure to return a value. So, if you defined `f` as above, then played with it in the shell,[1] you might get something like this:

```
>>> f
<function f at 0x82570>
>>> f(4)
5
>>> f(f(f(4)))
7
>>>
```

If we just evaluate `f`, Python tells us it's a function. Then we can apply it to 4 and get 5, or apply it multiple times, as shown.

What if we define

```
def g(x):
    x + 1
```

Now, when we play with it, we might get something like this:

```
>>> g(4)
>>> g(g(4))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in g
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>>
```

What happened!! First, when we evaluated `g(4)`, we got nothing at all, because our definition of `g` didn't return anything. Well...strictly speaking, it returned a special value called `None`, which the shell doesn't bother printing out. The value `None` has a special type, called `NoneType`. So, then, when we tried to apply `g` to the result of `g(4)`, it ended up trying to evaluate `g(None)`, which made it try to evaluate `None + 1`, which made it complain that it didn't know how to add something of type `NoneType` and something of type `int`.

Whenever you ask Python to do something it can't do, it will complain. You should learn to read the error messages, because they will give you valuable information about what's wrong with what you were asking for.

Print vs Return

Here are two different function definitions:

```
def f1(x):
    print x + 1
def f2(x):
    return x + 1
```

What happens when we call them?

```
>>> f1(3)
4
>>> f2(3)
4
```

It looks like they behave in exactly the same way. But they don't, really. Look at this example:

```
>>> print(f1(3))
4
None
>>> print(f2(3))
4
```

In the case of `f1`, the function, when evaluated, prints 4; then it returns the value `None`, which is printed by the Python shell. In the case of `f2`, it doesn't print anything, but it returns 4, which is printed by the Python shell. Finally, we can see the difference here:

```
>>> f1(3) + 1
4
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> f2(3) + 1
5
```

In the first case, the function doesn't return a value, so there's nothing to add to 1, and an error is generated. In the second case, the function returns the value 4, which is added to 1, and the result, 5, is printed by the Python read-eval-print loop.

The book *How to Think Like a Computer Scientist*, which we recommend reading, is translated from a version for Java, and it has a lot of `print` statements in it, to illustrate programming concepts. But for just about everything we do, it will be returned values that matter, and printing will be used only for debugging, or to give information to the user.

Footnotes:

[1] Although you can type procedure definitions directly into the shell, you won't want to work that way, because if there's a mistake in it, you'll have to type the whole thing in again. Instead, you should type your procedure definitions into a file, and then get Python to evaluate them. Look at the documentation for the programming environment we're using for an explanation of how to do that.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011