Stupid lambda tricks

This document describes some stupid tricks for using lambda expressions in Python (http://www.python.org). It is not intended for the "functional programming style" crowd, but for the "wacky geek fun" hackers.

If you have comments, or new ideas, please don't hesitate to contact me (mailto:not@p-nand-q.com).

Note: The new scoping rules for Python 2.2 are most useful for lambda expressions. If you haven't done so yet, do an update to 2.2 first.

Lambda limitations

To understand the need for such a page, the most common limitations on lambda expressions are:

- You can only use expressions, but not statements.
- You can only use one expression.
- You cannot declare and use local variables.

Adding Statements to Lambda

The official python documentation says, that you can only use expressions in lambda, but not statements. For example, the tutorial says in section 4.7.4: *"They (lambda expressions) are syntactically restricted to a single expression"*. You can

also see this from the syntax definition in section 5.10 of the "Python Reference Manual":

```
expression:     or_test | lambda_form
or_test:        and_test | or_test "or" and_test
and_test:       not_test | and_test "and" not_test
not_test:       comparison | "not" not_test
lambda_form:    "lambda" [parameter_list]: expression
```

Using *print* in lambda

The best solution is to define your own print function:

```
>>> out=lambda *x:sys.stdout.write(" ".join(map(str,x)))
>>> out("test",42)
test 42
```

Note that this function doesn't do string formatting - you can use the normal %-style syntax for that.

Using data assignment in lambda

In Python, assignments are statements (see "6.3 Assignment statements" in the "Python Reference Manual"), so you cannot easily use them in lambdas.

Data assignment for lists

To set an item in a list, you can use the member function *__setitem__*. For example, rather than writing

```
data[4] = 42
```

you can write

```
data.__setitem__(4,42)
```

Example: Here is a function that swaps two elements in a given list:

```
def swap(a,x,y):
    a[x] = (a[x], a[y])
    a[y] = a[x][0]
    a[x] = a[x][1]
```

You can write this as a lambda expression like this:

```
swap = lambda a,x,y:(lambda f=a.__setitem__:(f(x,(a[x],a[y])),
       f(y,a[x][0]),f(x,a[x][1])))()
```

Data assignment for locals

Sometimes you have local variables, and want to assign values to them. The best solution is to store all local variables in a list, and use list assignment as described above.

Using IF statements

Here is a quote from section "4.16." from "The Whole Python FAQ:"

> 4.16. Is there an equivalent of C's "?:" ternary operator?
>
> Not directly. In many cases you can mimic a?b:c with "a and b or c", but there's a flaw: if b is zero (or empty, or None -- anything that tests false) then c will be selected instead. In many cases you can prove by looking at the code that this can't happen (e.g. because b is a constant or has a type that can never be false), but in general this can be a problem. Tim Peters (who wishes it was Steve Majewski) suggested the following solution: (a and [b] or [c])[0]. Because [b] is a singleton list it is never false, so the wrong path is never taken; then applying [0] to the whole thing gets the b or c that you really wanted. Ugly, but it gets you there in the rare cases where it is really inconvenient to rewrite your code using 'if'.

This method is of course trivially defined as lambda:

```
IF = lambda a,b,c:(a() and [b()] or [c()])[0]
```

If you don't need else, you're even better off:

```
IF = lambda a,b:a() and b()
```

Using WHILE statements

Here is a generic while loop:

```
while expression():
    function()
```

So, given expression and function, you can write this as:

```
WHILE = lambda e,f:(e() and (f(),WHILE(e,f)) or 0)
```

Which is sort-of tail recursive, but uses a lot of stack. If e() and f() are fixed expressions, use

```
WHILE = lambda:e() and (f(),WHILE())
```

For one hell of a nice while-loop.

Using FOR statements

Here is a generic for loop:

```
for x in items:
    f(x)
```

Use map() instead. You can rewrite this as:

```
map(f,items)
```

Which has the benefit of being faster in most cases, too.

Using Multiple expressions in Lambda

There are several possible solutions to the problem of how to use several distinct expressions in one lambda statement. The general form of the problem is: How to use

```
    f1()
    f2()
    ...
    fn()
```

in one lambda statement. The basic idea used in most of the following is to rewrite that as

```
for function in (f1, f2, f3):
    function()
```

Use tuples and rely on evaluation order

If the functions are not related, you can say:

```
(f1(), f2(), ..., fn())
```

The current evaluation order is from left to right, so that should work.

Use tuples, and implement your own evaluation order

If you are a coward and fear that evaluation order will change in a future release of python, you can use *eval*

```
map(eval,("f1()","f2()",...,"fn()"))
```

You can also use *apply*

```
map(lambda x:apply(x,()),(f1,f2,f3))
```

Using local variables

The best idea is to define a helper lambda function that has the local variables as argument. For example, say you have this function:

```
def f(x):
    result = range(20)
    return do_something(result,x)
```

You can rewrite this as

```
f = lambda x: (lambda result=range(20):do_something(result,x))()
```

As mentioned above, local variables are best used in a list, because that way you can use *[].__setitem__* for data assignments.

Misc topics

Here are some other topics that don't seem to fit in any of the above.

Wrappers for list functions that return *None*

Many list functions return *None*, so you cannot easily use them in a stacked expression inside lambda. Examples of this are *[].sort()* and *[].reverse()*. Instead of writing:

```
list.some_function_that_returns_None()
function(list)
```

you can write

```
function((list.some_function_that_returns_None(),list)[1])
```

Another suggestion, by Dave Benjamin (http://www.ramenfest.com) is to use the interpretation of "None" as "False" in Python and write

```
function(list.some_function_that_returns_None() or list)
```

For instance, a simplified version of the dict() builtin can be implemented on earlier versions of Python like this:

```
>>> dict = lambda x: reduce(lambda x, y: x.update({y[0]:y[1]}) or x, x,
>>> dict({1: 2, 3: 4}.items())
{3: 4, 1: 2}
```

Here are a few more simple examples:

```
import random

sort = lambda x: (x.sort(),x)[1]
reverse  = lambda x: x.reverse() or x

test = range(20)
print test
random.shuffle(test)
print sort(test)
print reverse(sort(test))
```

The sort function mentioned in that sample code has the drawback of modifying the original list. Here are two solutions that return a sorted list while not modifying the original argument:

```
sort = lambda x: (lambda x: x.sort() or x)(x[:])
sort = lambda x: (lambda x=x[:]: x.sort() or x)()
```