1 RPN (Danny Yoo / Donovan Preston)

This code implements a reverse polish notation calculator. It’s deliberately written in a slightly weird way to avoid being taken used as a homework answer.

```python
# RPN calculator in Python.
# This version of RPN is meant to be incredibly silly. It abuses lambdas all around, uses global variables, and in general is just *grin* really silly. No one in in their right mind would do it this way.

stack = []

def rpn(commands):
    global stack
    stack = []
    for t in commands:
        t()
    assert len(stack) == 1
    return stack[0]

def makeOpCommand(op):
    return lambda: stack.append(op(stack.pop(), stack.pop()))

def makeCommands(line):
    ops = {'+': makeOpCommand(lambda x, y: x + y),
           '-': makeOpCommand(lambda y, x: x - y),
           '*': makeOpCommand(lambda x, y: x * y),
           '/': makeOpCommand(lambda y, x: x / y)}
    commands = []
    for t in line.split():
        # next line, using default parameters, is intentional.
        commands.append(ops.get(t, lambda t=t: stack.append(float(t))))
    return commands

if __name__ == '__main__':
    print(rpn(makeCommands('1 2.3 - 3 4 5 + * /')))```
Donovan Preston later modified this code to be even more obscene. In his words,

This version uses no statements. The entire program is a single expression. It is very side-effecty, though.

```python
globals().update(
    dict(defs=lambda **args: globals().update(args))),

defs(
    sys=__import__(‘sys’),
    operator=__import__(‘operator’)),

defs(
    swap=lambda op: lambda x, y: op(y, x),
    makeOp=lambda op: lambda stack: stack.append(op(stack.pop()),
    stack.pop())),

defs(
    ops=dict([(char, makeOp(op)) for (char, op) in
                [('+', operator.add), ('-', swap(operator.sub)),
                 ('*', operator.mul), ('/', swap(operator.div))]]),

defs(
    popStack=lambda _, stack: stack.pop(0),
    rpn=lambda commands, stack=[]: popStack([t(stack) for t in
                                      commands], stack),
    makeCommands=lambda line: [
        ops.get(t, lambda stack, t=t: stack.append(float(t)))
        for t in line.split()],

operator.eq(__name__, '__main__') and sys.stdout.write(
    str(rpn(makeCommands(‘1 2.3 - 3 4 5 + * /’)))+‘\n’)
```
Here is my cool code submission. It’s called the stan syntax, and it’s a very easy way to generate xml from python. My web framework, Nevow, includes a more complete implementation of this idea.

```python
***"stan constructs xml from pure python"***

class Proto(str):
    children = []
    proto = property(lambda self: self)
    def __call__(self, **kw):
        return Tag(self)(**kw)
    def __getitem__(self, *args):
        return Tag(self)[args]

class Tag(object):
    def __init__(self, proto):
        self.proto = proto
        self.attributes = {}
        self.children = []
    def __call__(self, **kw):
        self.attributes.update(kw)
        return self
    def __getitem__(self, *args):
        if not isinstance(args, (list, tuple)):
            args = [args]
        self.children.extend(args)
        return self

def flatten(it, indent=0):
    if isinstance(it, (Proto, Tag)):
        attributes = ', '.join(['='.join([key, '%s' % value]) for (key, value) in it.attributes.items()])
        if attributes:
            attributes = ' ' + attributes
        print ' ' * indent + ' <%s%s>' % (it.proto, attributes)
        for child in it.children:
            flatten(child, indent+1)
        print ' ' * indent + ' </%s>' % it.proto
    elif isinstance(it, (list, tuple)):
        [flatten(child, indent) for child in it]
    else:
        print ' ' * indent, it

html, head, title, body, h1, div, a, img = (}
Proto('html'), Proto('head'), Proto('title'), Proto('body'),
Proto('h1'), Proto('div'), Proto('a'), Proto('img'))

document = html[
    head[
        title['Hello'],
    ],
    body[
        h1['Goodbye'],
        div(style='color: red')['Red text!'],
        a(href='http://google.com/')[
            img(
                title='Link to google',
                src='http://google.com/images/logo.gif'),
                'Google'])
]

if __name__ == '__main__':
    flatten(document)
3 Python Metric Calculator (Reg Charney)

From Reg. Charney:

I should have a program for producing Python metrics. These measurements will include McCabe’s Cyclomatic Complexity, Fan In/Fan Out, Lines of Comments, Lines of Code, number of doc strings, etc. The unique thing for this type of program will be the output format.
4 Cookbook Recipes (Jimmy Retzlaff)

class attrdict(dict):
    """A dict whose items can also be accessed as member variables."
    
    >>> d = attrdict(a=1, b=2)
    >>> d['c'] = 3
    >>> print d.a, d.b, d.c
    1 2 3
    >>> d.b = 10
    >>> print d['b']
    10

    # but be careful, it's easy to hide methods
    >>> print d.get('c')
    3
    >>> d['get'] = 4
    >>> print d.get('a')
    Traceback (most recent call last):
    TypeError: 'int' object is not callable

    def __init__(self, *args, **kwargs):
        dict.__init__(self, *args, **kwargs)
        self.__dict__ = self

import BaseHTTPServer
import webbrowser

def LoadInDefaultBrowser(html):
    """Display html in the default web browser without creating a temp file.

    Instantiates a trivial http server and calls webbrowser.open with a URL
to retrieve html from that server.
    """

    class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
        def do_GET(self):
            bufferSize = 1024*1024
            for i in xrange(0, len(html), bufferSize):
                self.wfile.write(html[i:i+bufferSize])

            server = BaseHTTPServer.HTTPServer(('127.0.0.1', 0), RequestHandler)
            webbrowser.open('http://127.0.0.1:%s' % server.server_port)
            server.handle_request()

        if __name__ == '__main__':
            LoadInDefaultBrowser('<b>Hello World</b>')

def iterQueue(queue, sentinel):
    """Iterate over the values in queue until sentinel is reached.""
    while True:
        value = queue.get()
        if value != sentinel:
            yield value
        else:
            return
5 Pipe-like Syntax (Maxim Krikun)

Maxim Krikun’s Pipe-like syntax

```python
from itertools import izip, imap, count, ifilter
import re

def cat(fname):
    return file(fname).xreadlines()

class grep:
    """keep only lines that match the regexp""
    def __init__(self, pat, flags=0):
        self.fun = re.compile(pat, flags).match
    def __ror__(self, input):
        return ifilter(self.fun, input)

class tr:
    """apply arbitrary transform to each sequence element""
    def __init__(self, transform):
        self.tr = transform
    def __ror__(self, input):
        return imap(self.tr, input)

class printlines:
    """print sequence elements one per line""
    def __ror__(self, input):
        for l in input:
            print l

printlines=printlines()

class terminator:
    """to be used at the end of a pipe-sequence""
    def __init__(self, method):
        self.process = method
    def __ror__(self, input):
        return self.process(input)

# those objects transform generator to list, tuple or dict
aslist = terminator(list)
asdict = terminator(dict)
astuple = terminator(tuple)

# this object transforms seq to tuple sequence
enum = terminator(lambda input: izip(count(), input))

# example 1: equivalent to shell grep ".*/bin/bash" /etc/passwd
cat('/etc/passwd') | tr(str.rstrip) | grep('.*/bin/bash') | printlines

# example 2: get a list of int's methods beginning with '__r'
dir(int) | grep('__r') | aslist

# example 3: useless; returns a dict {0:'l',1:'a',2:'m',3:'b',4:'d',5:'a'}
'lambda' | enum | asdict
```