6. Functional Programming

6.1. Recursion

Defining solution of a problem in terms of the same problem, typically of smaller size, is called recursion. Recursion makes it possible to express solution of a problem very concisely and elegantly.

A function is called recursive if it makes call to itself. Typically, a recursive function will have a terminating condition and one or more recursive calls to itself.

6.1.1. Example: Computing Exponent

Mathematically we can define exponent of a number in terms of its smaller power.

```
def exp(x, n):
    """
    Computes the result of x raised to the power of n.

        >>> exp(2, 3)
        8
        >>> exp(3, 2)
        9
    """
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)
```

Lets look at the execution pattern.

```
exp(2, 4)
+-- 2 * exp(2, 3)
|       +-- 2 * exp(2, 2)
|       |       +-- 2 * exp(2, 1)
|       |       |       +-- 2 * exp(2, 0)
|       |       |       |       +-- 1
|       |       |       +-- 2 * 1
|       |       |       +-- 2
|       |       +-- 2 * 2
|       |       +-- 4
|       +-- 2 * 4
|       +-- 8
+-- 2 * 8
+-- 16
```

Number of calls to the above `exp` function is proportional to size of the problem, which is `n` here.

We can compute exponent in fewer steps if we use successive squaring.

```
def fast_exp(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_exp(x*x, n/2))
    else:
        return x * fast_exp(x, n-1)
```

Lets look at the execution pattern now.

```
fast_exp(2, 10)
+-- fast_exp(4, 5) # 2 * 2
|   +-- 4 * fast_exp(4, 4)
|   |       +-- fast_exp(16, 2) # 4 * 4
|   |       |   +-- fast_exp(256, 1) # 16 * 16
|   |       |   |   +-- 256 * fast_exp(256, 0)
|   |       |   |             +-- 1
|   |       |   |   +-- 256 * 1
|   |       |   |   +-- 256
|   |       |   +-- 256
|   |       +-- 256
|   +-- 4 * 256
|   +-- 1024
+-- 1024
1024
```

Problem 1: Implement a function `product` to multiply 2 numbers recursively using `+` and `-` operators only.

6.1.2. Example: Flatten a list

Supposed you have a nested list and want to flatten it.

```
def flatten_list(a, result=None):
    """Flattens a nested list.

        >>> flatten_list([ [1, 2, [3, 4] ], [5, 6], 7])
        [1, 2, 3, 4, 5, 6, 7]
    """
    if result is None:
        result = []

    for x in a:
        if isinstance(x, list):
            flatten_list(x, result)
        else:
            result.append(x)

    return result
```

Problem 2: Write a function `flatten_dict` to flatten a nested dictionary by joining the keys with `.` character.

```
>>> flatten_dict({'a': 1, 'b': {'x': 2, 'y': 3}, 'c': 4})
{'a': 1, 'b.x': 2, 'b.y': 3, 'c': 4}
```

Problem 3: Write a function `unflatten_dict` to do reverse of `flatten_dict`.

```
>>> unflatten_dict({'a': 1, 'b.x': 2, 'b.y': 3, 'c': 4})
{'a': 1, 'b': {'x': 2, 'y': 3}, 'c': 4}
```

Problem 4: Write a function `treemap` to map a function over nested list.

```
>>> treemap(lambda x: x*x, [1, 2, [3, 4, [5]]])
[1, 4, [9, 16, [25]]]
```

Problem 5: Write a function `tree_reverse` to reverse elements of a nested-list recursively.

```
>>> tree_reverse([[1, 2], [3, [4, 5]], 6])
[6, [[5, 4], 3], [2, 1]]
```

6.1.3. Example: JSON Encode

Lets look at more commonly used example of serializing a python datastructure into JSON (JavaScript Object Notation).

Here is an example of JSON record.

```
{
    "name": "Advanced Python Training",
    "date": "October 13, 2012",
    "completed": false,
    "instructor": {
        "name": "Anand Chitipothu",
        "website": "http://anandology.com/"
    },
    "participants": [
        {
            "name": "Participant 1",
            "email": "email1@example.com"
        },
        {
            "name": "Participant 2",
            "email": "email2@example.com"
        }
    ]
}
```

It looks very much like Python dictionaries and lists. There are some differences though. Strings are always enclosed in double quotes, booleans are represented as `true` and `false`.

The standard library module *json* provides functionality to work in JSON. Lets try to implement it now as it is very good example of use of recursion.

For simplicity, lets assume that strings will not have any special characters and can have space, tab and newline characters.

```
def json_encode(data):
    if isinstance(data, bool):
        if data:
            return "true"
        else:
            return "false"
    elif isinstance(data, (int, float)):
        return str(data)
    elif isinstance(data, str):
        return '"' + escape_string(data) + '"'
    elif isinstance(data, list):
        return "[" + ", ".join(json_encode(d) for d in data) + "]"
    else:
        raise TypeError("%s is not JSON serializable" % repr(data))

def escape_string(s):
    """Escapes double-quote, tab and new line characters in a string."""
    s = s.replace('"', '\\"')
    s = s.replace("\t", "\\t")
    s = s.replace("\n", "\\n")
    return s
```

This handles booleans, integers, strings, floats and lists, but doesn't handle dictionaries yet. That is left an exercise to the readers.

If you notice the block of code that is handling lists, we are calling *json_encode* recursively for each element of the list, that is required because each element can be of any type, even a list or a dictionary.

Problem 6: Complete the above implementation of `json_encode` by handling the case of dictionaries.

Problem 7: Implement a program `dirtree.py` that takes a directory as argument and prints all the files in that directory recursively as a tree.

Hint: Use `os.listdir` and `os.path.isdir` funtions.

```
$ python dirtree.py foo/
foo/
|-- a.txt
|-- b.txt
|-- bar/
|   |-- p.txt
|   `-- q.txt
`-- c.txt
```

Problem 8: Write a function `count_change` to count the number of ways to change any given amount. Available coins are also passed as argument to the function.

```
>>> count_change(10, [1, 5])
3
>>> count_change(10, [1, 2])
6
>>> count_change(100, [1, 5, 10, 25, 50])
292
```

Problem 9: Write a function `permute` to compute all possible permutations of elements of a given list.

```
>>> permute([1, 2, 3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

6.2. Higher Order Functions & Decorators

In Python, functions are first-class objects. They can be passed as arguments to other functions and a new functions can be returned from a function call.

6.2.1. Example: Tracing Function Calls

For example, consider the following `fib` function.

```
def fib(n):
    if n is 0 or n is 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Suppose we want to trace all the calls to the `fib` function. We can write a higher order function to return a new function, which prints whenever `fib` function is called.

```
def trace(f):
    def g(x):
        print f.__name__, x
        value = f(x)
        print 'return', repr(value)
        return value
    return g

fib = trace(fib)
print fib(3)
```

This produces the following output.

```
fib 3
fib 2
fib 1
return 1
fib 0
return 1
return 2
fib 1
return 1
return 3
3
```

Noticed that the trick here is at `fib = trace(fib)`. We have replaced the function `fib` with a new function, so whenever that function is called recursively, it is the our new function, which prints the trace before calling the orginal function.

To make the output more readable, let us indent the function calls.

```
def trace(f):
    f.indent = 0
    def g(x):
        print '|  ' * f.indent + '|--', f.__name__, x
        f.indent += 1
        value = f(x)
        print '|  ' * f.indent + '|--', 'return', repr(value)
        f.indent -= 1
        return value
    return g

fib = trace(fib)
print fib(4)
```

This produces the following output.

```
$ python fib.py
|-- fib 4
|   |-- fib 3
|   |   |-- fib 2
|   |   |   |-- fib 1
|   |   |   |   |-- return 1
|   |   |   |-- fib 0
|   |   |   |   |-- return 1
|   |   |   |-- return 2
|   |   |-- fib 1
|   |   |   |-- return 1
|   |   |-- return 3
|   |-- fib 2
|   |   |-- fib 1
|   |   |   |-- return 1
|   |   |-- fib 0
|   |   |   |-- return 1
|   |   |-- return 2
|   |-- return 5
5
```

This pattern is so useful that python has special syntax for specifying this concisely.

```
@trace
def fib(n):
    ...
```

It is equivalant of adding `fib = trace(fib)` after the function definition.

6.2.2. Example: Memoize

In the above example, it is clear that number of function calls are growing exponentially with the size of input and there is lot of redundant computation that is done.

Suppose we want to get rid of the redundant computation by caching the result of `fib` when it is called for the first time and reuse it when it is needed next time. Doing this is very popular in functional programming world and it is called `memoize`.

```
def memoize(f):
    cache = {}
    def g(x):
        if x not in cache:
            cache[x] = f(x)
        return cache[x]
    return g

fib = trace(fib)
fib = memoize(fib)
print fib(4)
```

If you notice, after `memoize`, growth of `fib` has become linear.

```
|-- fib 4
|  |-- fib 3
|  |  |-- fib 2
|  |  |  |-- fib 1
|  |  |  |  |-- return 1
|  |  |  |-- fib 0
|  |  |  |  |-- return 1
|  |  |  |-- return 2
|  |  |-- return 3
|  |-- return 5
5
```

Problem 10: Write a function `profile`, which takes a function as argument and returns a new function, which behaves exactly similar to the given function, except that it prints the time consumed in executing it.

```
>>> fib = profile(fib)
>>> fib(20)
time taken: 0.1 sec
10946
```

Problem 11: Write a function `vectorize` which takes a function `f` and return a new function, which takes a list as argument and calls `f` for every element and returns the result as a list.

```
>>> def square(x): return x * x
...
>>> f = vectorize(square)
>>> f([1, 2, 3])
[1, 4, 9]
>>> g = vectorize(len)
>>> g(["hello", "world"])
[5, 5]
>>> g([[1, 2], [2, 3, 4]])
[2, 3]
```

6.2.3. Example: unixcommand decorator

Many unix commands have a typical pattern. They accept multiple filenames as arguments, does some processing and prints the lines back. Some examples of such commands are `cat` and `grep`.

```
def unixcommand(f):
    def g(filenames):
        printlines(out for line in readlines(filenames)
                       for out in f(line))
    return g
```

Lets see how to use it.

```
@unixcommand
def cat(line):
    yield line

@unixcommand
def lowercase(line):
    yield line.lower()
```

6.3. exec & eval

Python privides the whole interpreter as a built-in function. You can pass a string and ask it is execute that piece of code at run time.

For example:

```
>>> exec("x = 1")
>>> x
1
```

By default `exec` works in the current environment, so it updated the globals in the above example. It is also possible to specify an environment to `exec`.

```
>>> env = {'a' : 42}
>>> exec('x = a+1', env)
>>> print env['x']
43
```

It is also possible to create functions or classes dynamically using `exec`, though it is usually not a good idea.

```
>>> code = 'def add_%d(x): return x + %d'
>>> for i in range(1, 5):
...     exec(code % (i, i))
...
>>> add_1(3)
4
>>> add_3(3)
6
```

`eval` is like `exec` but it takes an expression and returns its value.

```
>>> eval("2+3")
5
>>> a = 2
>>> eval("a * a")
4
>>> env = {'x' : 42}
>>> eval('x+1', env)
43
```