CSC326: Assignment 2-Solution

November 21, 2016

Problem 1. Function profiling is an important concept which helps improving application performance. The following should go to a file called **q1.py**. Create a function that decorates another function in the following way:

```
@profile
def foo ( ... ) :
    ...
```

What profile does is to record information about the runtime of, and the number of calls to the function it decorates using the clock function of the time module. For example:

```
from time import clock
start = clock()
# do something for a while
duration = clock() - start
```

Importantly, your implementation of profile should only record the runtime of a funtion if a global variable `PROFILE_FUNCTIONS` has the value `True`. Regardless of the value of `PROFILE_FUNCTIONS`, your decorator should not change the behavior of the function it decorates. That is, the decorated function should accept the same number and kinds of parameters and output the same values as if it were not decorated. Your decorator must also be generic, *i.e.* I should be able to decorate arbitrary functions with profile.

Finally, record the profiling results in a global dictionary `PROFILE_RESULTS`. The dictionary should map the profiled function names to the tuple: `(a, b)`, in which `a` is the average runtime over all calls to the function and `b` is the number of times the function is called.

Solution:

```
from time import clock

profile_results = {}
PROFILE_FUNCTIONS=1

def profile(func):
        def func_wrapper(*args, **kwargs):
            if PROFILE_FUNCTIONS:
                start = clock()
                ret = func(*args, **kwargs)
                duration = clock() - start
                if profile_results.get(func.__name__):
                    p     = profile_results[func.__name__]
                    new_d = (p[0]*p[1]+duration)/(p[1]+1)
                    new_f = p[1]+1
                    profile_results[func.__name__] = (new_d,new_f)
                else:
                    profile_results[func.__name__] = (duration,1)
                return ret
            else:
                return func(*args, **kwargs)
        return func_wrapper
```

Problem 2. The following will go into a file called **q2.py**. Using Functional Programming Style (without using any control statement like if-then-else, while loop), create a function named `find_product(l)` where `l` is a list of digits, *i.e.* integers from 0 to 9. The goal is to find the 5 consequtive elements of the list `l` that has the greatest product. The function should return a tuple, `(a,b)`, where `a` is the index of the greatest product and `b` is the greatest product value.

```
Example:
>>> find_product([1,2,3,4,5,6,4,2,1,3])
    (2, 1440)
```

Note: You are not allowed to use Python `max` builtin function.

Hint: Consider using enumerate for the index part

Solution:

```
def find_product(l):
    compare = lambda a,b: (a[1]<b[1] and b) or a
    single_mult = lambda a: reduce(lambda x,y:x*y, l[a:a+5])
    mult = map(single_mult, range(len(l)-4))
    return reduce(compare,list(enumerate(mult)))
```

Problem 3. The following will go into a file called **q3.py**. Implement your own version of `map`, `reduce`, and `filter` Python functions with the same semantics as the corresponding Python native functions. Name your functions as `my_map`, `my_reduce` and `my_filter`.

Solution:

```
def my_map(func,*args):
    max=0
    for arg in args:
        if max < len(arg):
            max=len(arg)
    for arg in args:
        arg = arg.extend([None] * (max-len(arg)))
    z = zip(*args)
    ret = []
    if func is None:
        return z
    else:
        for i in z:
            ret.append(func(*i))
        return ret

def my_reduce(func,iterable,init=None):
    if init!=None:
        ret = init
        for i in iterable:
            ret = func(ret,i)
    else:
        ret = iterable[0]
        for i in iterable[1:]:
            ret = func(ret,i)
    return ret

def my_filter(func,iterable):
    ret=[]
    for i in iterable:
        if func(i):
            ret.append(i)
    return ret
```

Problem 4. The following will go into a file called **q4.py**. Write an iterator which returns words from a big file, without reading the whole file into memory. Create function named `find_popular`, which accepts a text file name as parameter, and print out the list of the ten most popular words (by the number of occurrences).

Solution:

```
def iter_read_file(name):
    with open(name) as infile:
        for line in infile:
            yield line

def find_popular(name):
    dic = {}
    f = iter_read_file(name)
    for line in f:
        words = line.split()
        for word in words:
            if dic.get(word):
                dic[word] += 1
            else:
                dic[word] = 1
    t = sorted(dic.items(), key=lambda x:x[1],reverse=True)
    dic = dict(t[:10])
    dic = sorted(dic.items(), key=lambda x:x[1],reverse=True)
    for key in dic:
        print key[0]
```

Problem 5. The following will go into a file called **q5.py**. A sentence splitter is a program capable of splitting a text into sentences. The standard set of heuristics for sentence splitting includes (but isn't limited to) the following rules:

Sentence boundaries occur at one of "." (periods), "?" or "!", exccept that:

- Periods followed by whitespace followed by a lower case letter are not sentence boundaries.
- Periods followed by a digit with no intervening whitespace are not sentence boundaries.
- Periods followed by whitespace and then an upper case letter, but preceded by any of a short list of titles are not sentence boundaries. Sample titles include Mr., Mrs., Dr., and so on.
- Periods internal to a sequence of letters with no adjacent whitespace are not sentence boundaries (for example, www.aptex.com, or e.g).
- Periods followed by certain kinds of punctuation (notably comma and more periods) are probably not sentence boundaries.

Your task here is to write a function named split_sentence, that given the name of a text file is able to write its content with each sentence on a separate line.

Hint: Consider using iterators

Test your program with the following short text:

```
Mr. Smith bought cheapsite.com for 1.5 million dollars, i.e. he paid
a lot for it. Did he mind? Adam Jones Jr. thinks he didn't. In any
case, this isn't true... Well, with a probability of .9 it isn't.
```

The result should be:

```
Mr. Smith bought cheapsite.com for 1.5 million dollars, i.e. he paid a lot for it.
Did he mind?
Adam Jones Jr. thinks he didn't.
In any case, this isn't true...
Well, with a probability of .9 it isn't.
```

Solution:

```
def iter_read_sentence(name):
    with open(name) as infile:
        for line in infile:
            for c in line:
                yield c


def is_abbrv(s):
    l = len(s)
    return (    s[l-3:]=='Mr.'  or      s[l-3:]=='Dr.' or \
                s[l-4:]=='Mrs.'     or      s[l-3:]=='Jr.' or \
                s[l-3:]=='Ms.'      or      s[l-5:]=='Prof.' or \
                s[l-3:]=='Sr.'      or      s[l-3:]=='St.')

def split_sentence(name):
    t = iter_read_sentence(name)
    f = open('splited.txt','w')
    string_list = []
    string=''
    try:
        for c in t:
            string+=c
            if c=='.' or c=='?' or c=='!':
                n1 = t.next()
                if n1==' ':
                    l = len(string)
                    if is_abbrv(string):
                        string += n1
                    else:
                        n2 = t.next()
                        if n2>='a' and n2<='z':
                            string+=(n1+n2)
                        else:
                            string_list.append(string+'\n')
                            string=n2
                else:
                    string+=n1
    except StopIteration:
        string_list.append(string)
    f.writelines(string_list)
```