PE03: 09/01/2021 — Solutions

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2020/2021

An example of solutions for the 5 questions in this Practical on computer evaluation.

1. sWAP cASE

Write a Python function `swap_case(astr)` that, for a given string `astr`, returns another string with all lowercase letters converted to uppercase letters and vice-versa.

Solution:

```
def swap_case(astr):
    # solve it with (old fashion) iteration
    result = ""
    for letter in astr:
        if letter.isupper():
            result += letter.lower()
        else:
            result += letter.upper()
    return result

# alternative solution 1
def swap_case2(astr):
    # it's worth knowing the methods of the string object
    return astr.swapcase()

# alternative solution 2
def swap_case3(astr):
    # solve it with the focus on data (as in PE04)
    return ''.join([l.lower() if l.isupper() else l.upper() for l in astr])
```

2. Dictionary unique values

Write a Python function `unique_values(alist)` that, given a list of dictionaries, returns a set with the values of all dictionaries.

Solution:

```
def unique_values(alist):
    # solve it with (old fashion) iteration
    uniqs = []
    for d in alist:
        for v in d.values():
            if v not in uniqs:
                uniqs += [v]
    return set(uniqs)

# alternative solution 1
def unique_values2(alist):
    # using a set to remove duplicates
    uniqs = set()
    for d in alist:
        for v in d.values():
            uniqs.add(v)
    return uniqs

# alternative solution 2
def unique_values3(alist):
    # solve it with the focus on data (as in PE04)
    return set(value for adict in alist for value in adict.values())
```

3. Count elements

Write a Python function `rec_count(alist)` that computes the frequency of each element in the nested list `alist` (i.e., a list which may contain lists, which themselves may contain other lists, and so on), and returns it in the form of a dictionary.

Solution:

```
def rec_count(alist):
    # using an hybrid solution with iteration and recursion
    result = {}
    for el in alist:
        # see if the element is a singleton or a list
        if type(el) != list:
            d = {el: 1}
        else:
            d = rec_count(el)
        # compact version:
        # d = rec_count(el) if type(el) == list else {el: 1}

        # update the dictionary after unwinding the recursive calls
        for k, v in d.items():
            result[k] = result.get(k, 0) + v
    return result

# alternative solution
def rec_count2(alist):
    # using an auxiliary function with an accumulator
    d = {}
    count_aux(alist, d)
    return d

def count_aux(alist, d):
    # base case
    if alist == []:
        return []
    # recursive calls
    if type(alist[0]) == list:
        # the head is a list, count the head and count the tail
        return count_aux(alist[0], d) + count_aux(alist[1:], d)
    # else, the head is a singleton, add it to the dictionary
    d[alist[0]] = d.get(alist[0], 0) + 1
    return count_aux(alist[1:], d)
```

4. Nested filter/map/reduce

Write a Python function `nested_fmr(f, m, r, lst)` that, given 3 functions `f`, `m` and `r`, recursively applies the filter **f**, followed by the map **m**, followed by the reduce with **r** on every sublist of the nested list **lst.**

For example with `f = lambda x: x<10`, m = `lambda x: abs(x)` and `r = lambda x,y: x+y`, and the nested list `[[4,-20],[15,-1],-4,3]`, the function should return `8`, with the intermediary step `[24,1,-4,3]`, after applying filter/map/reduce to the nested list.

Solution:

```
# importing functools for reduce()
import functools

def nested_fmr(f, m, r, lst):
    # using recursion to deal with the nested list
    # base case
    if type(lst) != list:
        return lst
    # iteration with the recursive call
    aux = []
    for l in lst:
        aux.append(nested_fmr(f, m, r, l))
    return functools.reduce(r, map(m, filter(f, aux)))

# alternative solution
def nested_fmr2(f, m, r, lst):
    # solve it with the focus on data (as in PE04)
    if type(lst) != list:
        return lst
    aux = [nested_fmr2(f, m, r, l) for l in lst]
    return functools.reduce(r, map(m, filter(f, aux)))
```

5. Calculator with fractions

Write a function `calculator(expr)` that given an expression `expr` computes its value. The expression `expr` may be a fraction tuple as `(numerator, denominator)` or it may be a tuple composed of `(expr, operator, expr)`. The valid operators are only multiplication (`'*'`) and division (`'/'`) and all values are non-negative. Simplify the result to the smallest integer dividend.

For example, `calculator(((1, 3), '*', (2, 5)))` calculates the expression 1/3 * 2/5 and evaluates to the fraction 2/15, represented by `(2, 15)`.

Solution:

```
def gcd(x, y):
    # math.gcd()
    while(y):
        x, y = y, x % y
    return x

def calculator_rec(expr):
    # the base case
    if len(expr) == 2:
        return expr
    # the recursive calls
    e1, op, e2 = expr
    e1 = calculator(e1)
    e2 = calculator(e2)
    if op == '*':
        return (e1[0]*e2[0], e1[1]*e2[1])
    else:  # '/'
        return (e1[0]*e2[1], e1[1]*e2[0])

def calculator(expr):
    # the result numerator and denominator
    num, den = calculator_rec(expr)
    # the greatest common divisor of the two integers
    d = gcd(num, den)
    # simplify the fraction
    return (num // d, den // d)
```

The end.