PE05: 22/02/2021 — Solutions

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

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

1. Argmax

Write a Python function `argmax(lst)` that receives a non-empty list `lst` of integers and returns the index of the first instance of the maximum value.

Solution:

```python
def argmax(lst):
    max_value = max(lst)
    for i in range(len(lst)):
        if lst[i] == max_value:
            break
    return i

# alternative solution 1
def argmax2(lst):
    max_value = -999999
    for i in range(len(lst)):
        if lst[i] > max_value:
            first = i
            max_value = lst[i]
    return first

# alternative solution 2
def argmax3(lst):
    return lst.index(max(lst))
```

2. Sum sublists

Write a Python function `sum_sublists(lst)` that sum the numbers in the same index of all sublists of `lst`. You can assume all sublists have the same size.

For example, `sum_sublists([[1, 2, 0], [2, 5, 0], [0, 1, 2]])` will return `[3, 8, 2]`.

Solution:

```
def sum_sublists(lst):
    if len(lst) == 0:
        return []
    result = lst[0].copy()
    for sublist in lst[1:]:
        for i in range(len(sublist)):
            result[i] += sublist[i]
    return result
```

3. Generator square root

Consider a `k`-bounded version of the *Babylonian method* to calculate the square root of a number `num`:

1. Make an initial guess, x_0=`num/2`
2. Find a better guess by applying formula x_{n+1}`:=(`x_n`+num/`x_n`)/2`
3. If the process converged (the digits of x_{n+1} and x_n agree on 4 decimal places), terminate
4. If too many steps have been performed (`n>k`), terminate
5. Otherwise, go to 2

Write a generator function `sqrt(num, k)` that iteratively *yields* all guesses (rounded to 2 digits) of the method above until termination.

Solution:

```
def sqrt(num, k):
    aprox = num / 2
    for _ in range(0, k):
        yield round(aprox, 2)
        prev = aprox
        aprox = (aprox + num / aprox) / 2
        if round(prev, 4) == round(aprox, 4):
            break
```

4. n-lists

Write a Python function `n_lists(alist, n)` that receives a list `alist` containing sub-lists (which can themselves recursively contain sub-lists) and an integer `n` (≥ 1), and returns a list such that each sub-list has length ≤ `n` elements. You must divide each sub-list into several sub-lists, as necessary.

For example, if `alist=[[1, 2, 3], [5, 6]]` and `n=2`, then the result should be `[[1, 2], [3], [5, 6]]`. That is, the list `[1, 2, 3]` is split so that each sub-list only has at most two elements (because `n=2`).

Solution:

```
def n_lists(alist, n):
    result = []
    # iterate over the sublists and use recursion for each
    for l in alist:
        if type(l) == int:
            result.append(l)                        # base case
        else:
            for i in range(0, len(l), n):
                split = l[i:i+n]                    # split
                result.append(n_lists(split, n))  # recursive case
    return result
```

5. Balanced parenthesis

Write a Python function `balanced_parenthesis(expression)` that receives an arithmetic `expression` (as a string) containing zero or more parenthesis — `'('`, `')'`, `'['` and `']'` and returns the number of matched pair of parentheses if they are balanced or `-1` otherwise (the parentheses are not balanced).

Note: the last parenthesis to be open, `'('`, `'['`, is the first parenthesis to be closed `')'`, `']'`.

For example:

1. For `expression="(2 + 3 * 5 / 78 + [23 / 34 - 89] * 3)"` the function returns `2` because the expression has 2 balanced pairs of parentheses.
2. For `expression="(2 + [3 * 5 / 78 + [23 / 34 - (89 * 3)])]"` the function returns `-1` because not all parentheses are balanced.

Solution:

```
UNBALANCED = -1

def balanced_parenthesis(expression):
    matched = 0
    stack = []
    for ch in expression:
        if ch in ('(', '['):
            # keep a record of an open parenthesis
            stack.append(ch)
        elif ch in (')', ']'):
            # there must be an open parenthesis to match the close one
            ch2 = '(' if ch == ')' else '['
            if len(stack) == 0 or stack.pop() != ch2:
                # there's no open parenthesis to match
                return UNBALANCED
            # another pair of matched parenthesis
            matched += 1
    # true if there's no open parenthesis left to match
    return matched if len(stack) == 0 else UNBALANCED
```

Alternative solution:

```
UNBALANCED = -1

close_parenthesis = {'(': ')', '[': ']'}
opposite_parenthesis = {'(': '[', '[': '('}

def balanced_parenthesis2(expression):
    matched = 0
    # remove all characters except parentheses
    expression = ''.join(c for c in expression if c in ['(', ')', '[', ']'])
    while len(expression):
        matched += 1
        ch = expression[0]
        if ch == ')' or ch == ']':
            return UNBALANCED
        # find the parenthesis that terminates this one and remove both
        found = False
        count_this = 1
        count_other = 0
        for i in range(1, len(expression)):
            if expression[i] == ch:
                count_this += 1
            elif expression[i] == close_parenthesis[ch]:
                count_this -= 1
            elif expression[i] == opposite_parenthesis[ch]:
                count_other += 1
            elif expression[i] == close_parenthesis[opposite_parenthesis[ch]]:
                count_other -= 1
            if count_this == 0:
                found = True
                expression = expression[1:i] + expression[i+1:]
                break
        if count_other > 0 or not found:
            return UNBALANCED
    return matched
```

The end.