PE06: 12/04/2021 — 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. Where is the minimum

Write a Python function `find_min(numbers)` that receives a non-empty tuple of integers `numbers` and returns the index of the first element of the minimum value.

Solution:

```
def find_min(numbers):
    min_value = min(numbers)
    for i, n in enumerate(numbers):
        if n == min_value:
            return i

# alternative solution
def find_min2(numbers):
    min_value = numbers[0]
    pos = 0
    for i in range(1, len(numbers)):
        if numbers[i] < min_value:
            min_value = numbers[i]
            pos = i
    return pos

# alternative solution
def find_min3(numbers):
      return numbers.index(min(numbers))
```

2. Differences

Write a Python function `diff(lst)` that receives a list `lst` of numbers and returns a tuple with the difference between consecutive elements of the given list, or an empty tuple when the list has less than 2 elements.

For example, `diff([1, 4, 3])` should return `(-3, 1)`, because `1-4=-3` and `4-3=1`.

Solution:

```
def diff(t):
    ret = ()
    for i in range(len(t)-1):
        ret += (t[i] - t[i+1],)
    return ret
```

3. Binary numbers generator

Write a generator function `binary_range(start, end)` that receives two integers, `start` and `end`, and iteratively *yields* each binary number in the interval [`start`, `end`[.

Solution:

```
def binary_range(start, end):
    assert start <= end, "start <= end"
    b = start
    while b != end:
        yield b
        if b % 10 == 0:
            b += 1
        else:
            i = 0
            while b % 10 == 1:
                i += 1
                b //= 10
            b = (b+1) * 10**i

# alternative solution
def binary_range2(start, end):
    b = str(start)
    while b != str(end):
        yield int(b)
        if b[-1] == '0':
            b = b[:-1] + '1'
        else:
            b = '1' + '0'*len(b)

# alternative solution
def binary_range3(start, end):
    i = 0
    while i < int(str(end), 2):
        if i >= int(str(start), 2):
            yield int(bin(i)[2:])
        i += 1
```

4. Interleave Lists

Write a Python function `interleave(alist1, alist2)` that, given two lists of the same shape, `alist1` and `alist2`, which may contain recursive lists, returns a flattened (non-recursive) list with the interleaved elements, the first from `alist1` and the second from `alist2`.

Solution:

```
# recursive solution
def interleave(l1, l2):
    if type(l1) != list: # base condition
        return [l1, l2]
    if len(l1) == 0:     # or len(l2) == 0
        return []
    return interleave(l1[0], l2[0]) + interleave(l1[1:], l2[1:])

# another solution with iteration and recursion
def interleave2(l1, l2):
    if type(l1) != list: # base condition
        return [l1, l2]
    res = []
    for l1i, l2i in zip(l1, l2):
        res += interleave2(l1i, l2i)
    return res

# another solution with iteration and recursion
def interleave3(l1, l2):
    res = []
    for i in range(len(l1)):
        if type(l1[i]) != list:
            res += [l1[i], l2[i]]
        else:
            res += interleave3(l1[i], l2[i])
    return res
```

5. Additions before multiplications

Write a function `normalize(expr)` that given an expression `expr` , normalizes it so that additions come before multiplications, using the associative laws:

1. (a+b)*c ⇨ (a*c)+(b*c)
2. c*(a+b) ⇨ (c*a)+(c*b)
3. (a+b)*(c+d) ⇨ ((a*c)+(a*d))+((b*c)+(b*d))

The expression `expr` is either an integer or a tuple composed of `(expr, operator, expr)`. The only valid operators are addition (`'+'`) and multiplication (`'*'`).

Solution:

```
def normalize(expr):
    if type(expr) == int:
        return expr
    e1, op, e2 = expr
    e1 = normalize(e1)
    e2 = normalize(e2)
    if op == '*':
        # Rule 1: (a+b)*c => (a*c)+(b*c)
        if type(e1) == tuple and (type(e2) == int or e2[1] == '*'):
            if e1[1] == '+':
                a, _, b = e1
                c = e2
                # must not forget to re-normalize
                return normalize(((a, '*', c), '+', (b, '*', c)))
        # Rule 2: c*(a+b) => (c*a)+(c*b)
        if (type(e1) == int or e1[1] == '*') and type(e2) == tuple:
            if e2[1] == '+':
                c = e1
                a, _, b = e2
                return normalize(((c, '*', a), '+', (c, '*', b)))
        # Rule 3: (a+b)*(c+d) => ((a*c)+(a*d))+((b*c)+(b*d))
        if type(e1) == tuple and type(e2) == tuple:
            if e1[1] == '+' and e2[1] == '+':
                a, _, b = e1
                c, _, d = e2
                return normalize((((a, '*', c), '+', (a, '*', d)), '+', ((b,
'*', c), '+', (b, '*', d))))
    # all other cases are already normalized
    return (e1, op, e2)
```

The end.