PE04: 29/01/2021 — 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. Lists to dictionary

Write a Python function `lists_to_dict(list1, list2)` that, given two lists, returns a dictionary with keys from `list1` and the corresponding values from `list2`, in the given order.

Solution:

```
def lists_to_dict(list1, list2):
    # there's no repetitions in list1
    return dict(zip(list1,list2))


# alternative solution 2
def lists_to_dict2(list1, list2):
    return {list1[i]: list2[i] for i in range(len(list1))}


# alternative solution 3
def lists_to_dict3(list1, list2):
    res = {}
    for i, k in enumerate(list1):
        res[k] = list2[i]
    return res
```

2. Sum dictionaries

Write a Python function **`sum_dicts(lst)`** that receives a list **`lst`** of dictionaries, each one having a set of keys and corresponding integer values.

The function returns a single dictionary with all the keys available in the input dictionaries. Values associated with the same key are added up in the final dictionary.

For example, if **`lst=[{'a': 5, 'b': 3}, {'a': 1, 'c': 0}]`** then the result is **`{'a': 6, 'b': 3, 'c': 0}`**.

Solution:

```
def sum_dicts(lst):
    result = {}
    for d in lst:
        for k, v in d.items():
            # add the value v to 0 or to the value for an existing key
            result[k] = result.get(k, 0) + v
    return result


# alternative solution using reduce()
import functools

def sum_dicts2(lst):
    # function to merge 2 dictionaries
    merge = lambda d1, d2: \
            {k: d1.get(k, 0) + d2.get(k, 0) \
                for k in set(list(d1.keys()) + list(d2.keys()))}
    return functools.reduce(merge, lst, {})
```

3. Fibonacci generator

Write a generator function **`fib(start, end)`** which generates the Fibonacci sequence of integer numbers starting at the i-th number **`start >= 1`** and finishing at the i-th number **`end >= start`**.

The numbers of the sequence are calculated using the following formula: the first two numbers of the sequence are equal to 1 and each consecutive number is the sum of the last two numbers.

For example, if **`start=1`** and **`end=7`** then the *yielded* sequence is **`[1, 1, 2, 3, 5, 8, 13]`**.

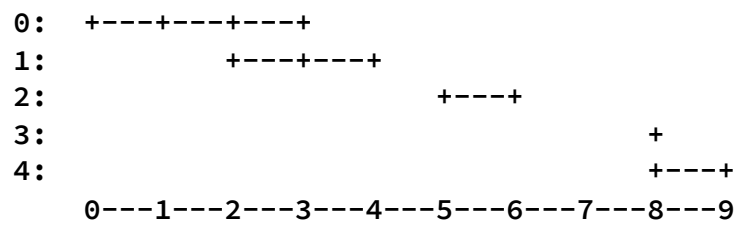
Solution:

```
def fib(start, end):
    i = 1
    a, b = 1, 1
    # must calculate the sequence from the beginning independently of start
    while i <= end:
        if i >= start:
            yield a
        a, b = b, a + b
        i += 1
```

4. Overlap segments

Write a Python function **`overlaps(segments)`** that receives a list of **`segments`**, where each segment is represented by a tuple **`(start, end)`**, containing the start and end points of the segment, and returns a **set of tuples** with the indices of the overlapping segments.

For example, in this illustration,



`overlaps([(0, 3), (2, 4), (5, 6), (8, 8), (8, 9)])`, the function should return **`{(0, 1), (3, 4)}`**.

Solution:

```
def is_overlap(s1, s2) -> bool:
    """ find out if the two segments (start, end) overlap """
    return not (s1[0] > s2[1] or s1[1] < s2[0])

def overlaps(segments):
    # using a set comprehension, compare each segment with all the others
    return {(i, j) for i in range(len(segments))
                       for j in range(i+1, len(segments))
                                  if is_overlap(segments[i], segments[j])}
```

5. Process commands

Write a Python function `rec_hof(hofs, lst)` that recursively applies a list of higher-order functions `hofs` to a nested list `lst`. Each element of `hofs` is a tuple with a "function operation" (for example, `map`) and a "function argument" (for example, a lambda function).

For example, the list `hofs=[(map, sum), (filter, lambda x: x>0)]` applied to `lst=[[-1, 2],[2, 3]]` should return `[2, 5]`, by applying the `filter` with the lambda to the sub-lists (resulting in `[[2],[2, 3]]`) and then the `map` of function `sum` to the top-list.

Inputs are assumed to be always valid.

Solution:

```
def rec_hof(hofs, lst):
    if hofs == []:
        # base case, there's no more functions to apply
        return lst
    # get the transformed sub-lists
    aux = [rec_hof(hofs[1:], x) for x in lst]
    # apply the first higher-order function to the sub-lists
    (f_op, f_arg) = hofs[0]
    return f_op(f_arg, aux)


# alternative solution 2
def rec_hof2(hofs, lst):
    if hofs == []:
        # base case, there's no more functions to apply
        return lst
    # get the transformed sub-lists
    aux = []
    for l in lst:
        aux.append(rec_hof(hofs[1:], l))
    # apply the first higher-order function to the sub-lists
    (f_op, f_arg) = hofs[0]
    return f_op(f_arg, aux)


# alternative solution 3
import functools
def rec_hof3(hofs, lst):
    if hofs == []:
        # base case, there's no more functions to apply
        return lst
    # get the transformed sub-lists using partial function application
    aux = map(functools.partial(rec_hof, hofs[1:]), lst)
    # apply the first higher-order function to the sub-lists
    (f_op, f_arg) = hofs[0]
    return f_op(f_arg, aux)
```

The end.