NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5001—Software Development Fundamentals
Academic Year 2023/2024, Semester 2
PROBLEM SET 3

SEQUENCES

These exercises are optional and ungraded, and for your own practice only. Contact `yongqi@nus.edu.sg` for queries.

SIMPLE EXERCISES

Question 1. Without using IDLE, determine the output from the following print statements.

```
tup_a = (10, 11, 12, 13)
tup_b = ('IT', 5001)
tup_c = tup_a + tup_b
print('1.', 11 in tup_a)
print('2.', 14 in tup_b)
print('3.', 'I' in tup_c)
print('4.', tup_b[1])
tup_d = tup_b[0] * 4
print('5.', tup_d)
print('6.', tup_b[1] * 4)
tup_e = tup_d[1:]
print('7.', tup_e)
tup_f = tup_d[::-1]
print('8.', tup_f)
tup_g = tup_d[1:-1:2]
print('9.', tup_g)
tup_h = tup_d[-1:6:-2]
print('10.', tup_h)
tup_i = (1)
print('11.', tup_i)
print('12.', tup_i * 4)
tup_j = (1,)
print('13.', tup_j)
print('14.', tup_j * 4)
print('15.', min(tup_a))
print('16.', max(tup_a))
print('17.', min(tup_c))
print('18.', max(tup_c))
print('19.', min(tup_e))
print('20.', max(tup_e))
```

```
print('21.', (10, 12) < tup_a)
```

Question 2. Without using IDLE, determine the output from the following print statements.

```
lst_a = ['IT', 5001]
lst_b = ['E', ('is', 'easy')]
lst_c = lst_a + lst_b
print('1.', lst_a)
print('2.', lst_b)
print('3.', lst_c)
tup_a = ('IT', 5001)
tup_a[1] = 5003
lst_a[1] = 5003
print('4.', lst_a)
lst_a.append('E')
print('5.', lst_a)
lst_a.extend('easy')
print('6.', lst_a)
cpy_b = lst_b[:]
print('7.', cpy_b)
cpy_b[1] = 'is hard'
print('8.', cpy_b)
print('9.', lst_b)
lst_d = [1, [2], 3]
cpy_d = lst_d[:]
print('10.', cpy_d)
print('11.', lst_d)
lst_d[1][0] = 9
print('12.', cpy_d)
print('13.', lst_d)
print('14.', lst_d == cpy_d)
print('15.', lst_d is cpy_d)
print('16.', lst_d[1] == cpy_d[1])
print('17.', lst_d[1] is cpy_d[1])
```

Question 3. Consider `f`.

```
def f(ls):
    if not ls:
        return 1
    return ls[0] * f(ls[1:])
```

Question 3 (i). What does it do?

Question 3 (ii). We know that slicing entails copying, and thus this solution might not be optimal. Propose a new `f` that does the same, is still recursive, but overcomes this inefficiency.

Question 4. Consider this program fragment:

```
>>> def f(ls = []):
...     return ls
>>> a = f()
>>> a.append(1)
>>> a
[1]
>>> b = f()
>>> b
[1]
```

What does this tell you about default parameter values? Propose a change to `f` such that if the user omits the argument `ls` when calling `f`, within `f`, `ls` is a new empty list.

Question 5. Try running the following in the console.

```
>>> a = [0] * 3
>>> b = [[0]] * 3
```

Then, mess around with the two lists. What do you observe?

SHORT PROGRAMMING EXERCISES

Question 6. Write a function `is_all_lower_alphabet` that takes in a string and returns `True` if all characters in the string are lowercase alphabets, `False` otherwise. Use `any` or `all`.

Question 7. Write a function `r` that takes in a list of numbers, and returns the equivalent list where consecutive duplicate numbers are removed. Write one iterative and one recursive version. Here is an example run:

```
>>> a = [1, 1, 2, 2, 2, 1, 1, 3, 3, 4]
>>> r(a)
[1, 2, 1, 3, 4]
>>> a
[1, 1, 2, 2, 2, 1, 1, 3, 3, 4]
```

Question 8. Suppose you are given the following helper functions:

```
def head(tp):
    return tp[0]
def tail(tp):
```

```
    return tp[1:]
def cons(el, tp):
    return (el,) + tp
def empty(tp):
    return not tp
```

Write a function `concat` that concatenates two tuples recursively. You are only to use the helper functions given to you without using any of the sequence operations.

Question 9. We have seen the `deep_copy` function in class that supports deeply copying a list that contains a reference to itself

```
def deep_copy(x):
    if isinstance(x, int):
        return x
    res = []
    for i in x:
        if i is x:
            res.append(res)
        else:
            res.append(deep_copy(i))
    return res
```

Write a function that can safely deep-copy any kind of multi-dimensional list of integers (including a list `z` who contains list `y` such that `y` contains a reference to `z`).

Question 10. *"The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves."* [Wikipedia].

Write a function `evolve` that receives a *configuration*—a two-dimensional list of 0s (the cell is dead) and 1s (the cell is live)—and performs one round of evolution on the configuration based on the following rules:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Note that a cell a is a neighbour of another cell b if a is directly to the left, right, above, or below (and not diagonally) b.

LONG PROGRAMMING QUESTION

Question 11. We are going to continue running our own burger shop! Recall that we were able to get the price of a burger using the following functions (or something to that effect):

```
def ingredient_price(ing):
    prices = 50, 80, 150, 70, 40, 90
    ingredients = 'BCPVOM'
    return prices[ingredients.index(ing)]
def burger_price(burger):
    return sum(ingredient_price(i) for i in burger)
```

This time, we are going to create a mini Point of Sales system! Here are the features of our system:

1. add an item to order (can be ala carte burgers or meals)
2. change an order
3. remove an order
4. view orders
5. undo

Here's an example run of a customer using this system.

```
>>> add BVPB
>>> add BVPCVB L
>>> view
==== YOUR ORDER ====
0. BVPB: $3.20
1. BVPCVB (L meal): $7.70
Total: $10.90
====================
>>> change 0 BVPB S
>>> view
==== YOUR ORDER ====
0. BVPB (S meal): $4.20
1. BVPCVB (L meal): $7.70
Total: $11.90
====================
>>> remove 1
>>> view
==== YOUR ORDER ====
0. BVPB (S meal): $4.20
Total: $4.20
====================
>>> undo
```

```
>>> view
==== YOUR ORDER ====
0. BVPB (S meal): $4.20
1. BVPCVB (L meal): $7.70
Total: $11.90
====================
>>> done
```

Question 11 (i). The first step of creating such a complex system is to design it at a high level of abstraction. Before this, we define our terms.

burger. A burger (such as `'BVPB'`).

meal. A meal consisting of some burger and of some size. For example, there can be a meal consisting of `'BVPB'` and of size L. S stands for Small ($1), M stands for Medium ($2), and L stands for Large ($3). The price of the meal is simply the price of the burger plus the size of the meal. So, given that the price of `'BVPB'` is $3.20, the price of a medium `'BVPB'` meal would be $5.20.

item. Either a burger or meal.

order. A group of items that the user wants to order.

Think about what data structures should represent burgers, meals, items and orders, the algorithms you want to use for the system etc.

Question 11 (ii). Write a function `add_item(order, item)` that adds `item` to `order`.

Question 11 (iii). Write a function `remove_item(order, index)` that adds removes the item at index `index` from `order`.

Question 11 (iv). Write a function `change_item(order, index, new_item)` that changes the item at index `index` in `order` into `new_item`.

Question 11 (v). Write a function `print_order(order)` that prints each item in the order along with its index and price, and at the end, print the total price of the order. You may refer to format shown in the example run above for inspiration.

Question 11 (vi). Now that we are able to tie it altogether, go ahead and complete the system. If you had chosen the right data structures for each part of the system, this should not be difficult. Otherwise, you might have to rework some of the functions that you have defined earlier.

SOLUTIONS

Question 1. Finding out the solutions to these is quite simple—just write the expressions in IDLE to see the results after evaluating them.

Some notable expressions are:

(1.3) `'I'` is not in `tup_c`, even though the string `'IT'` is.

(1.11) `tup_i` is really just the number 1. That is because `(1)` is taken as a parenthesized expression, not a tuple containing 1.

(1.17) and (1.18) Both of these expressions give an error because the elements within `tup_c` cannot be compared.

(1.21) Strings, tuples and lists follow the same comparison rules of lexicographical ordering. Ranges do not support inequality comparisons.

Question 2. Finding out the solutions to these is quite simple—just write the expressions in IDLE to see the results after evaluating them.

Some notable expressions are:

(After 2.3) The assignment statement `tup_a[1] = 5003` is obviously incorrect because tuples do not support item assignment.

(2.5) The `append` method accepts a single element and appends it to the list.

(2.6) The `extend` method accepts an iterable and appends each element of the iterable to the list. Remember that strings are sequences, and therefore iterables, so we are really appending each character in `'easy'` one at a time.

(2.8) and (2.9) `cpy_b` is a shallow copy of `lst_b`, so amending `cpy_b` itself will not affect `lst_b`.

(2.12) and (2.13) `cpy_d` is a shallow copy of `lst_d`, but `cpy_d[1]` and `lst_d[1]` refer to the exact same list, therefore any amendments to it will be visible by both `cpy_d` and `lst_d`

(2.14) to (2.17) `==` is simply an equality check; in the context of lists, it simply asks if the two lists have equal contents. `is` asks if both lists are the exact same object; in 2.15 we know that `cpy_d` and `lst_d` are separate lists, but in 2.17 `cpy_d[1]` and `lst_d[1]` refer to the exact same list.

Question 3.

i. `f` multiplies each element in `ls` together. More specifically, suppose $ls = [e_1, e_2, ..., e_n]$, then $f(ls) = e_1 \times (e_2 \times (... \times (e_n \times 1)))$.

ii. Instead of slicing, we can just pass in the index which marks the beginning of the list in consideration. Your solution may look something like the following:

```
def f(ls, i = 0):
    if i >= len(ls):
        return 1
    return ls[i] * f(ls, i + 1)
```

Question 4. Default parameter values are only evaluated once—this means that all calls to `f` where the `ls` argument is not supplied will return the exact same list. One quick solution is to use something called a sentinel value—some 'dummy' value to denote that something special is to happen when it is encountered. For example, we can use `None`.

```
def f(ls = None):
    if ls is None:
        ls = []
    return ls
```

Question 5. We observe that repetition repeats the contents of the list. Therefore, suppose `[[0]]` was really `[0x2]` and the reference `0x2` was referring to `[0]`, `b` really takes on the list `[0x2, 0x2, 0x2]`. If you are not convinced, try the following:

```
>>> a = [0] * 3
>>> a[0] = 1
>>> a
[1, 0, 0]
>>> b = [[0]] * 3
>>> b[0][0] = 1
>>> b
[[1], [1], [1]]
```

This is a common mistake that beginners make when trying to construct a 2D list of zeroes. Usually, we intend to do the following:

```
>>> b = [[0] for i in range(3)]
>>> b
[[0], [0], [0]]
>>> b[0][0] = 1
>>> b
[[1], [0], [0]]
```

Question 6. The idea is, if all of the characters in the string are lowercase alphabets, we return True. Otherwise, we return False. To check whether a character is a lowercase alphabet, we can see if its unicode code point is between 97 (a) and 122 (z). Thus, your solution might look something like the following:

```python
def is_all_lower_alphabet(s):
    return all(97 <= ord(char) <= 122 for char in s)
```

Question 7. This question is tricky. We'll start with the iterative version.

One idea is to build the new desired list from scratch. We start that list empty (let's call it `res`), and for each element in the input list, if that element is not equal to `res[-1]`, we add it to `res`. Otherwise, we do not.

Your solution might look something like this:

```python
def r(ls):
    res = []
    for i in ls:
        if not res or i != res[-1]:
            res.append(i)
    return res
```

Note that if `res` is empty, we immediately append the element to it; this is to prevent performing `res[-1]` when `res` is empty which will result in an `IndexError`.

Now we come to the recursive version. For `r(ls)`, we simply return `ls` if the length of `ls` is 1 or less. Otherwise, we think of a smaller problem from `r(ls)` and assume that `r` returns the correct result for that problem. We let that smaller problem be `r(ls[1:])` and assume that the result is correct. We then think about how we can use `r(ls[1:])` to solve `r(ls)`.

Actually, we just need to determine whether `ls[0]` should be included in the result. Let `res` = `r(ls[1:])`. If `ls[0] == res[0]`, then we just return `res`. Otherwise, we leave `ls[0]` in the solution for `r(ls)` by returning `[ls[0]] + res`. This leads us to the following relation for `r`:

$$\texttt{r(ls)} = \begin{cases} \texttt{ls}, & \texttt{len(ls) <= 1} \\ \texttt{r(ls[1:])}, & \texttt{ls[0] == r(ls[1:])[0]} \\ \texttt{[ls[0]] + r(ls[1:])}, & \text{otherwise} \end{cases}$$

Writing code for the above relation is trivial.

```python
def r(ls):
    if len(ls) <= 1:
        return ls
    res = r(ls[1:])
    if ls[0] == res[0]:
        return res
    return [ls[0]] + res
```

Question 8. This question is hard. `head` obtains the first element of the tuple, `tail` obtains the slice of the tuple starting from the second element, and `cons` prepends an element into the tuple. For any tuple `t`, `t = cons(head(t), tail(t))`.

The idea is as follows. For some `concat(a, b)`, if `a` is empty, we simply return `b`. Otherwise, we think of a smaller problem and assume that `concat` returns the correct result for that problem. We let the smaller problem from `concat(a, b)` be `concat(tail(a), b)` (logically, this is `concat(a[1:], b)`).

Let `r` be `concat(tail(a), b)`. Suppose we assume that `r` is the correct concatenation of `a[1:]` to `b`. To use `r` to solve `concat(a, b)`, we simply prepend `a[0]` back to `r`, i.e. we return `cons(head(a), r)`.

Thus, we come up with the following relation for `concat`:

$$\texttt{concat(a, b)} = \begin{cases} \texttt{b}, & \texttt{empty(a)} \\ \texttt{cons(head(a), concat(tail(a), b))}, & \text{otherwise} \end{cases}$$

Writing code for the above relation is trivial.

```
def concat(a, b):
    if empty(a):
        return b
    return cons(head(a), concat(tail(a), b))
```

Question 9. The key idea is that we can maintain a list of pairs, where each pair's first element is the list to be copied, and the second element is the result after copying. If a list to deep copy has already been computed before and is in the cache, then we return that result.

```
def deep_copy(x):
    # use an auxilliary function that receives the
    # cache
    def aux(x, cache):
        # base case
        if isinstance(x, int): return x
        # another base case; if the result of x has
        # been computed before and is found in the cache,
        # return the previously computed result
        for p, q in cache:
            # use is to perform reference checks
            if p is x:
                return q
        # standard accumulator pattern
        res = []
        # add the result list to the cache first
        cache.append((x, res))
        for i in x:
            res.append(aux(i, cache))
```



```
        return res
    # let the cache be a mutable list
    return aux(x, [])
```

Those familiar with dictionaries (which we will learn in week 4) can use it instead.

```
def deep_copy(x):
    # use an auxilliary function that receives the
    # cache
    def aux(x, cache):
        # base case
        if isinstance(x, int): return x
        # use id(x) to obtain the reference to x
        if id(x) in cache: return cache[id(x)]
        cache[id(x)] = res = []
        res.extend(aux(i, cache) for i in x)
        return res
    # let the cache be a mutable dictionary
    return aux(x, {})
```

This is a technique known as *memoization* which is typically used for a different purpose: dynamic programming.

Question 10. We can start by writing a function that receives the configuration and some coordinates, and determines the number of live neighbours of the cell at that coordinates.

```
def num_live_neighbours(config, r, c):
    # function to determine if cell (x, y) exists
    def is_valid_cell(x, y):
        return 0 <= x < len(config) and 0 <= y < len(config[x])
    # all possible neighbours
    possible_neighbours = [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]
    # get sum of values of neighbours
    return sum(config[i][j]
               for i, j in possible_neighbours
               if is_valid_cell(i, j))
```

Then solving the problem is relatively straightforward; simply use the accumulator pattern to produce the resulting list. Remember not to produce any side-effects. Solution below is self-explanatory.

```python
def evolve(config):
    # "deep copy" (actually we are initializing the 2D list with
    # zeroes
    # standard accumulator pattern
    res = [[0] * len(row) for row in config]
    # iterate over coordinates (we need them)
    for i in range(len(config)):
        for j in range(len(config[i])):
            if config[i][j]:
                # live condition
                if 2 <= num_live_neighbours(config, i, j) <= 3:
                    res[i][j] = 1
            else:
                # dead condition
                if num_live_neighbours(config, i, j) == 3:
                    res[i][j] = 1
    return res
```

Question 11 (i). This is subjective and typically based on client/organization requirements, but for our case we can simply go with our intuition and say that a meal is a **tuple** where the first element is a burger and the second element is the string denoting the size of the meal. An item is then a **tuple** either consisting only a burger (string), or it is a meal (tuple). An order will be a **list** of tuples. Any other reasonable choice is okay.

Question 11 (ii). We are going to avoid side-effects on our orders (even though they are mutable lists) since 1) it is generally good practice and 2) we should avoid mutating orders to keep track of the history of orders.

```python
def add_item(order, item):
    """
    Adds an item to an order.
    """
    return order + [item]
```

Question 11 (iii).

```python
def remove_item(order, index):
    """
    Removes an item from an order given its index.
    """
    order = order[:]
    order.pop(index)
    return order
```

Question 11 (iv).

```
def change_item(order, index, new_item):
    """
    Replaces an item in an order to a new item given
    its index.
    """
    order = order[:]
    order[index] = new_item
    return order
```

Question 11 (v). We should have a function that receives an item and computes its price, for convenience's sake.

```
def item_price(item):
    """
    Obtains the price of a single item in an order.
    """
    if len(item) == 1:
        return burger_price(item[0])
    return burger_price(item[0]) + ('SML'.index(item[1]) + 1) * 100
```

Then write our solution:

```
def print_order(order):
    """
    Prints an order.
    """
    print('==== YOUR ORDER ====')
    order_total = 0
    for i in range(len(order)):
        item = order[i]
        price = item_price(item)
        order_total += price
        if len(item) == 1:
            print(f'{i}. {item[0]}: ${price // 100}.{price % 100}')
        else:
            print(f'{i}. {item[0]} ({item[1]} meal): ${price // 100}.{price % 100}')
    print(f'Total: ${order_total // 100}.{order_total % 100}')
    print('====================')
```

Question 11 (vi). Because we kept our solution side-effect free, no changes need to be made to our solutions from earlier. We simply let the order history be a list of orders, and go from there.

```
def main():
    history = [[]]
    while True:
        command = input('>>> ').split(' ')
        if command[0] == 'add':
            history.append(add_item(history[-1], tuple(command[1:])))
        if command[0] == 'view':
            print_order(history[-1])
        if command[0] == 'change':
            history.append(change_item(history[-1], int(command[1]), tuple(command[2:])))
        if command[0] == 'undo':
            history.pop()
        if command[0] == 'remove':
            history.append(remove_item(history[-1], int(command[1])))
        if command[0] == 'done':
            break

if __name__ == '__main__':
    main()
```

– End of Problem Set 3 –