Exercises

Set 4

DM857 Introduction to Programming
DS830 Introduction to Programming

Lists

In this set you will gain practice with slicing, list comprehension, list concatenation and repetition, and some common methods of `list`.

1. For each of the following programs, compute its output without running them.

   (a)
   ```
   xs = [0,1,2,3,4,5]
   print(xs[0],xs[2],x[-1])
   ```
   (b)
   ```
   xs = [0,1,2,3,4,5]
   print(xs[1:2])
   print(xs[1:3])
   print(xs[:3])
   print(xs[3:])
   print(xs[1:4:2])
   print(xs[1:5:2])
   print(xs[1:15:2])
   print(xs[15:1:2])
   ```
   (c)
   ```
   xs = [[0,1],[2,3,4],[],[5]]
   print(xs[1])
   print(xs[1][1])
   print(xs[:3][1])
   print(xs[1][1:])
   ```
   (d)
   ```
   xs = [0,1,2,3,4,5]
   print(xs[len(xs)])
   ```

2. For each of the following programs, compute its output without running them.

   (a)
   ```
   xs = [0,1,2,3,4,5]
   xs[0] = xs[2]
   print(xs[0],xs[2])
   ```
   (b)
   ```
   xs = [0,1,2,3,4,5]
   del xs[2]
   print(xs)
   ```
   (c)
   ```
   xs = [0,1,2,3,4,5]
   xs.append(7)
   print(xs)
   ```
   (d)
   ```
   xs = [0,1,2,3,4,5]
   print(xs.pop())
   print(xs)
   ```

Exercises 3–5 focus on slicing, 7–28 on list comprehension, 29–38 on building lists with recursion. Solve the exercises in the remainder without using loops.

3. Define a function `odd_positions(xs:list)->list` that returns a list with all elements of `xs` in odd positions.

4. Define a function `even_positions(xs:list)->list` that returns a list with all elements of `xs` in even positions.

5. Define a function `reverse(xs:list)->list` that returns a list with all elements of `xs` in reverse order without using the the built-in function `reversed` or method `list.reverse`.

6. Define a function `runs(pattern:list,length:int)->list` that returns a list of the given length filled with `length` elements of `xs`. For instance, `runs([0, 1, 2], 0)` returns `[]` and `runs([0, 1, 2], 7)` returns `[0, 1, 2, 0, 1, 2, 0]`.

7. Define a function `doubles(xs:List[float])->List[float]` that returns a list with all elements of `xs` multiplied by 2.

8. Define a function `squares(xs:List[float])->List[float]` that returns a list with all elements of `xs` raised to the power of 2.

9. Define a function `even_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are even.

10. Define a function `odd_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are odd.

11. Define a function `smaller_than_filter(xs:List[int],cutoff:int)->List[int]` that returns a list with all elements of `xs` smaller than the given cut-off value.

12. Define a function `greater_than_filter(xs:List[int],cutoff:int)->List[int]` that returns a list with all elements of `xs` greater than cut-off.

13. Define a function `squares_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are square numbers.

14. Define a function `remove_all(xs:List[int],v:int)->List[int]` that returns a list with all elements of `xs` that are not equal `v` without using `list.remove` or `del`.

15. Define a function `count(xs:List[int],v:int)->int` that returns the number of elements of `xs` equal to `v` without using `list.count` or `list.index`.

16. Define a function `count_any(xs:List[int],vs:List[int])->int` that returns the number of elements of `xs` that are equal to any element of `vs` without using `list.count` or `list.index`.

17. Define a function `has_any(xs:List[int],vs:List[int])->bool` that checks if `xs` has at least one element equal to any element of `vs` without using `list.count` or `list.index`.

18. Define a function `has_none(xs:List[int],vs:List[int])->int` that checks if `xs` has no element equal to any element of `vs` without using `list.count` or `list.index`.

19. Define a function `has_all(xs:List[int],vs:List[int])->int` that checks if for every element of `vs` there is an element in `xs` equal to it without using `list.count` or `list.index`.

20. Define a function `unique_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are unique in `xs`, i.e., those elements that are not equal to any other element of `xs` without using `list.count` or `list.index`.

21. Define a function `repeated_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are not unique in `xs`, i.e., those elements that are equal to some other element of `xs` without using `list.count` or `list.index`.

22. Define a function `replace(xs:List[int],a:int,b:int)->List[int]` that returns a new list with the elements of `xs` except for those equal to `a` which are replaced with `b`.

23. Define a function `lowpass(xs:List[float],cutoff:float)->List[float]` that returns a list with all elements of `xs` replacing those larger than the cutoff value with it.

24. Define a function `highpass(xs:List[float],cutoff:float)->List[float]` that returns a list with all elements of `xs` replacing those smaller than the cutoff value with it.

25. Define a function `odd_in_odd(xs:List[int])->List[int]` that returns a list with all odd elements that occur in odd positions in xs.

26. Define a function `gcds(xs:List[int], ys:List[int])->List[int]` that returns a list with the greatest common divisor for each pair of elements of `xs` and ys. Recall from Set 2, Exercise 16, that you can compute the greatest common divisor for m and n (both positive) using Euclides' algorithm:

$$gcd(m,n) = \begin{cases} m & \text{if } m = n \\ gcd(m, n-m) & \text{if } m < n \\ gcd(m-n, n) & \text{if } m > n \end{cases}$$

27. Define a function `count_maximal(xs:List[int])->int` that returns the number of maximal elements of `xs`.[1]

28. Define a function `count_minimal(xs:List[int])->int` that returns the number of maximal elements of `xs`.[1]

[1] Approach `count_maximal` and `count_minimal` as exercises about list comprehension for the sake of gaining practice, we will see more efficient solutions using recursion or loops.