Exercises

Set 7

DM536 Introduction to Programming
DM562 Scientific Programming
DM857 Introduction to Programming
DS830 Introduction to Programming

1 Programming with loops and lists

1. Write a function `is_subset(l1,l2)` that checks if for every element of `l1` there is an equal element in `l2`.
2. Write a function `is_subset_id(l1,l2)` that checks if for every element of `l1` there is an identical element in `l2`.
3. Write a function `is_sorted(l)` that checks if the given list is sorted (without using any sorting function).
4. Write a function `is_sorted_reverse(l)` that checks if the given list is sorted in reverse order (without using any sorting or reversing function).
5. Write a function `first_index_max(l)` that returns the index of the first occurrence of the maximum element in `l`.
6. Write a function `last_index_max(l)` that returns the index of the first occurrence of the maximum element in `l`.
7. Write a function `double_it(l)` that takes a list of numbers `l` and replaces every number in it with its double.
8. Write a function `square_it(l)` that takes a list of numbers `l` and replaces every number in it with its square.
9. Write a function `parity(l)` that replaces each element in `l` by `0`, if it is even, or `1` if it is odd.
10. Write a function `apply_it(l,f)` that takes a list `l` and a function `f` with one argument and replaces every element of `l` with the result of `f` applied to it. Using this function, define a function `round_it(l)` that given a list of floating point numbers rounds them.
11. Write a function `below_and_above(l,n)` that returns a list with two elements: the first is the count of elements in `l` smaller than `n`, the second is the count of elements in `l` bigger than `n`.
12. Write a function `perfect_shuffle(l1,l2)` that takes two lists and returns a list constructed by taking one element from each list (assume `l1` and `l2` have the same length).
13. Write a function `longest_increasing_sequence(l)` that returns the length of the longest increasing sequence of elements in `l`.

14. The sieve of Eratosthenes is one of the oldest algorithms to find all prime numbers up to a given n. First, one writes down a list containing all numbers from 1 to n, and crosses out the 1. Next, one picks the next number k from the list that has not been crossed out, and crosses out all larger multiples of k. When the end of the list is reached, the numbers not crossed out are precisely the prime numbers smaller than or equal to n. Write a function `eratosthenes(n)` that returns the list of prime numbers smaller than `n` and uses Eratosthenes' algorithm to compute it. (Hint: use a list of n booleans to remember if a number is crossed-out, be careful with indexes as they start from 0.)

2 Programming with loops and nested lists

In the following, `m` is a list of lists.

1. Write a function `print_lengths(m)` that prints the length of each list in `m`.
2. Write a function `print_rows(m)` that prints each list in `m` on a separate line.
3. Write a function `max_length(m)` that returns the length of the longest list in `m`.
4. Write a function `total_length(m)` that returns the combined length of list in `m`.
5. Write a function `sum_2d(m)` that takes a list of lists of numbers (e.g., `[[1, 2], [3, 4], [5]]`) and returns the sum of all its elements.
6. Write a function `count_2d(m,c)` that returns the number of occurrences of `c` in `m` (without creating intermediate lists).
7. Write a function `max_2d(m)` that returns the maximum element in `m` (without creating intermediate lists).
8. Write a function `increment_2d(m)` that increments every number in `m` by one. For instance `increment_2d([[1, 2], [], [3]])` should return `[[2,3], [], [4]]`.
9. Write a function `parity_2d(m)` that replaces each element in `m` by `0` if even and by `1` if odd.
10. Write a function `chunks(l,n)` that takes a list `l` and returns a list of its "chunks" by breaking `l` in lists of length `n` (the last chunk can be shorter if there are not enough elements).

```
>>> chunks([1, 2, 3, 4, 5, 6, 7, 8, 9], 4)
[[1, 2, 3, 4], [5, 6, 7, 8], [9]]
```

11. Write a function `exact_chunks(l,n)` that behaves like `chunk(l,n)` except that chunks must have exactly length `n` for a total of `len(l) // n` chunks (extra elements are ignored).

```
>>> exact_chunks([1, 2, 3, 4, 5, 6, 7, 8, 9], 4)
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

12. Write a function `dealing(l,n)` that takes a list `l` and returns a list of `n` lists obtained by distributing the elements of `l` in rounds (like dealing cards to players one at a time) until there are no more elements to distribute.

```
>>> dealing([1, 2, 3, 4, 5, 6, 7, 8, 9], 4)
[[1, 5, 9], [2, 6], [3, 7], [4, 8]]
```

13. Write a function `exact_dealing(l,n)` that behaves like `dealing(l,n)` except that the sublists must have the same length and extra elements are ignored.

```
>>> exact_dealing([1, 2, 3, 4, 5, 6, 7, 8, 9], 4)
[[1, 5], [2, 6], [3, 7], [4, 8]]
```

14. Write a function `differences(l)` that takes a list of numbers `l` and returns list of lists such that: its first line is `l`; and each other line contains the differences between consecutive elements of the previous lines.

```
>>> differences([2, 1, 5, -2])
[[2, 1, 5, -2], [1, -4, 7], [5, -11], [16]]
```

15. Write a function `pascal(n)` that returns the first `n` lines of Pascal's triangle: its first line is `[1]`, and every other line contains a `1`, followed by the sums of all consecutive pairs of elements of the previous line, and a `1` at the end. For example, `pascal(4)` should return `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]`.

16. Write a function `trim_ends(m,w)` that trims every list in `m` to have at most `w` elements by deleting indexes at the end.

```
>>> trim_ends([[1, 2, 3], ['a', 'b', 'c', 'd'], [-1, -2, -3], ['e']], 2)
[[1, 2], ['a', 'b'], [-1, -2], ['e']]
```

17. Write a function `fill_ends(m,d)` that fill every list in `m` with `d` such that they all have the same length.

```
>>> fill_ends([[1, 2], ['a', 'b', 'c'], [], ['d']], '?')
[[1, 2, '?'], ['a', 'b', 'c'], ['?', '?', '?'], ['e', '?', '?']]
```

3 Programming with loops and matrices

In the following, `m` is a list of lists.

1. Write a function `is_matrix(m)` that checks whether `m` represents a matrix (i.e., if all inner lists have the same length).

2. Write a function `is_square_matrix(m)` that checks whether `m` represents a square matrix (i.e., the inner lists have the same length as `m`).

3. Write a function `scalar_sum(m,s)` that takes a matrix `m` and a number `s` and increments each element of `m` by `s`.

4. Write a function `scalar_prod(m,s)` that takes a matrix `m` and a number `s` and multiplies each element of `m` by `s`.

5. Write a function `print_matrix(m)` that prints `m` aligning its elements in columns.

```
>>> print_matrix([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]])
1  2  3  4
5  6  7  8
9 10 11 12
```

6. Write a function `print_table(m)` that prints `m` aligning its elements in columns using '|', '-', '+' to draw lines.

```
>>> print_table([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
+---+----+----+----+
| 1 |  2 |  3 |  4 |
+---+----+----+----+
| 5 |  6 |  7 |  8 |
+---+----+----+----+
| 9 | 10 | 11 | 12 |
+---+----+----+----+
```

(To make prettier tables, you can use box-drawing characters `https://en.wikipedia.org/wiki/Box-drawing_character`. You can copy-paste the necessary characters or refer to them by their unicode number e.g., `print(u'\u250C')` prints a top-left corner.)

7. Write a function `zeros(n)` that returns a matrix with `n` rows and `n` columns whose entries are all zeros.

8. Write a function `identity(n)` that returns a matrix a matrix with `n` rows and `n` columns whose entries are `1` in the diagonal and `0` otherwise.

```
>>> identity(3)
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> print_matrix(identity(3))
1 0 0
0 1 0
0 0 1
```

9. Write functions `del_col(m,j)` and `del_row(m,i)` that delete the `j`-th column and `i`-th row from the given matrix `m`.

```
>>> del_col([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], 0)
[[2, 3, 4], [6, 7, 8], [10, 11, 12]]
>>> del_row([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], 0)
[[5, 6, 7, 8], [9, 10, 11, 12]]
```

10. Write a function `transposed_square(m)` that takes a square matrix `m` and returns a new matrix obtained by flipping `m` over the diagonal (so the element at i,j in `m` ends up in j,i in the new matrix).

```
>>> transposed_square([[1, 2], [3, 4]])
[[1, 3], [2, 4]]

>>> transposed_square([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

11. Write a function `transpose_this(m)` that takes a square matrix `m` flips it over the diagonal (without creating a new matrix).

12. Write a function `transposed(m)` that takes a matrix `m` and returns its transposed matrix.

```
>>> transposed([[1, 2, 3], [4, 5, 6]])
[[1, 4], [2, 5], [3, 6]]
```