Exercises

Set 5

DM857 Introduction to Programming
DS830 Introduction to Programming

1 Programming with loops and lists

1. Define an iterative function `print_all(xs:list)` that prints all elements of `xs`, one per line.
2. Write a function `increment(xs:List[int])` that increments every number in `xs` by one.
3. Define an iterative function `double(xs:List[float])` that replaces every number in `xs` with its double.
4. Define an iterative function `square(l:List[float])` that replaces every number in `xs` with its square.
5. Define an iterative function `reverse(l:list)` that takes a list and reverses it in place (without invoking `list.reverse`).
6. Define an iterative function `parity(l)` that replaces each element in `l` by `0`, if it is even, or `1` if it is odd.
7. 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.
8. Define an iterative function `even_filter(xs:List[int])->List[int]` that returns a list with all elements of `xs` that are even.
9. Define an iterative function `odd_filter(xs:List[int])->List[int]` that returns a new list with all elements of `xs` that are odd.
10. Define an iterative function `smaller_than_filter(xs:List[int],cutoff:int)->List[int]` that returns a new list with all elements of `xs` smaller than the given cut-off value.
11. Define an iterative function `greater_than_filter(xs:List[int],cutoff:int)->List[int]` that returns a new list with all elements of `xs` greater than cut-off.
12. Define an iterative function `squares_filter(xs:List[int])->List[int]` that returns a new list with all elements of `xs` that are square numbers.
13. Define an iterative function `remove_all(xs:List[int],v:int)->List[int]` that returns a new list with all elements of `xs` that are not equal `v`.
14. Define an iterative function `sum(xs:List[float])->float` that returns the sum of a list of numbers.
15. Define an iterative function `mul(xs:List[float])->float` that returns the product of a list of numbers.

16. Define an iterative function `avg(xs:List[float])->float` that returns the arithmetic mean of a list of numbers.

17. Define an iterative function `maximum(xs:List[float])->float` that returns the maximum in a list of numbers.

18. Define an iterative function `minimum(xs:List[float])->float` that returns the minimum in a list of numbers.

19. Define an iterative function `first_index_max(xs:List[float])->int` that returns the index of the first occurrence of the maximum element in `xs`.

20. Define an iterative function `last_index_max(xs:List[float])->int` that returns the index of the last occurrence of the maximum element in `xs`.

21. Define an iterative function `count(xs:list,v:Any)->int` that returns the number of elements in `xs` equal to `v`.

22. Define an iterative function `count_id(xs:list,v:Any)->int` that returns the number of elements in `xs` identical to `v`.

23. Define an iterative function `count_any(xs:list,vs:list)->int` that returns the number of occurrences of elements of `vs` in `xs`.

24. Define an iterative function `count_maximal(xs:List[int])->int` that returns the number of maximal elements in `xs` (with a single pass on the list).

25. Define an iterative function `count_minimal(xs:List[int])->int` that returns the number of minimal elements in `xs` (with a single pass on the list).

26. Define an iterative function `count_not_maximal(xs:List[int])->int` that returns the number of elements of `xs` smaller than its maximum (with a single pass on the list).

27. Define an iterative function `count_not_minimal(xs:List[int])->int` that returns the number of elements of `xs` larger than its minimum (with a single pass on the list).

28. Define an iterative function `unique(xs:List[int])->List[int]` that returns a new list with all elements of `xs` that are unique in `xs` (with a single pass of `xs`).

29. Define an iterative function `repeated(xs:List[int])->List[int]` that returns a new list with all elements of `xs` that are not unique in `xs` (with a single pass of `xs`).

30. Define an iterative function `scale_to(xs:List[int],a:int,b:int)->List[int]` that returns a new list with all elements of `xs` scaled to the interval $[a, b]$.

31. Define an iterative function `hbars(xs:List[int])->None` that prints a textual horizontal bar chart representing the values in `xs`. Initially, you may assume that the values in `xs` are not negative.

```
>>> hbars([2,1,5,0])
    0 |==
    1 |=
    2 |=====
    3 |
```

Extend your initial solution to accept also negative values.

```
>>> hbars([2,0,4,-1])
    0  |==
    1  |
    2  |====
    3 =|
```

32. Extend your solution from the previous exercise to accept an optional argument `width` and scale the graph to fit the given width (in characters).

```
>>> hbars([2,0,4,-2],18)
    0     |========
    1     |
    2     |================
    3 ====|
```

33. Define an iterative function `comp_table(xs:List[int],ys:List[int])->None` that prints a `len(xs)`-by-`len(ys)` matrix of characters where a character in position `i`,`j` is `'+'` if `xs[i] > ys[j]`, `'-'` if `xs[i] < ys[j]`, and `' '` otherwise.

```
>>> comp_table([2,1,5,0,3,2,1],[0,1,3,2])
    +++ +++
    + +-++
    --+- --
     -+-+ -
```

34. Define an iterative function `is_subset(xs:list,ys:list)->bool` that checks if for each element of `xs` there is an element in `xs` equal to it.

35. Define an iterative function `is_sorted(xs:List[int])->bool` that checks if `xs` is sorted (without using any sorting function).

36. Define an iterative function `is_sorted_reverse(xs:list[int])->bool` that checks if the given list is sorted in reverse order (without using any sorting or reversing function).

37. Define an iterative function `perfect_shuffle(xs:list,ys:list)->list` that takes two lists and returns a list constructed by taking one element from each list (assume `xs` and `ys` have the same length).

38. Define an iterative function `longest_increasing_sequence(xs:List[int])->int` that returns the length of the longest increasing sequence of elements in `xs`.

39. 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. Define a function `eratosthenes(n>int)` 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

1. Define an iterative function `print_up_triangle(n:int)->None` that prints an upside "right triangle" with base and height `n` and made of asterisks like the one below.

```
>>> print_up_triangle(3)
*
**
***
```

2. Define an iterative function `print_down_triangle(n:int)->None` that prints a downside "right triangle" with base and height `n:int` and made of asterisks like the one below.

```
>>> print_down_triangle(3)
***
**
*
```

3. Define an iterative function `print_iso_triangle(n:int)->None` that prints an upside isosceles triangle made of asterisks like the one below.

```
>>> print_iso_triangle(3)
    *
   ***
  *****
```

4. Define an iterative function `factorial(n:int)->int` that returns $n!$, the factorial of n ($n! = 1 \cdot 2 \cdot \ldots \cdot n$).

5. Define an iterative function `factorial_sequence(n:int)->List[int]` that returns a list with $[0!, 1!, \ldots, n!]$.

6. Define an iterative function `double_factorial(n:int)->int` that returns $n!!$ ($n!! = 1 \cdot 3 \cdot 5 \cdot \ldots \cdot n$ if n is odd and $n!! = 2 \cdot 4 \cdot 6 \cdot \ldots \cdot n$ if n is even).

7. Define an iterative function `sum_up_to(n:int)->int` that that returns the sum of all natural numbers smaller than or equal to `n`.

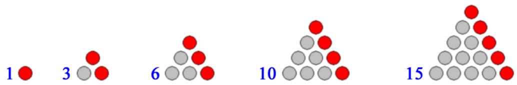
8. Define an iterative function `sum_between(m:int,n:int)->` that returns the sum of all integer numbers greater than `m` and smaller than `n`.

9. Define an iterative function `sum_even_between(m:int,n:int)->int` that returns the sum of all integer even numbers greater than `m` and smaller than `n`.

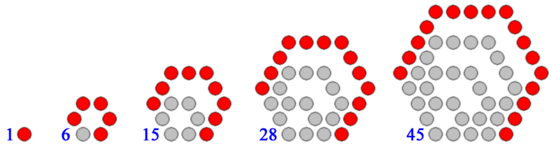
10. Define an iterative function `sum_odds_between(m:int,n:int)->int` that returns the sum of all integer odd numbers greater than `m` and smaller than `n`.

11. Define an iterative function `divisors(n:int)->List[int]` that given a positive integer `n` returns the list of all integers that divide `n`.

12. Define an iterative function `triangular(n:int)->int` that returns the `n`-th triangular number. A number is triangular if it counts objects that can be arranged in an regular triangle.



13. Define an iterative function `hexagonal(n:int)->int` that returns the `n`-th hexagonal number. A number is hexagonal if it counts objects that can be arranged in an regular hexagon.



14. Suppose that f is a continuous and positive function over an interval $[a, b]$. The area between axis and the graph of f in the interval $[a, b]$ (also called the integral of f in $[a, b]$) can be computed as precisely as required by the following method: we divide the interval $[a, b]$ in n subintervals of equal width, and approximate the integral of f in each subinterval by the area of the rectangle whose height is given by the value of f value in the midpoint of the interval. Define a function `integrate(f:Callable\[[float],float\],a:float,b:float,n:int)->float` that given a function `f(x:float)->float` [1], floats `a` and `b`, and a positive integer `n`, returns the approximate value of the integral of `f` over [a,b] using the algorithm above.

15. Define a function `is_prime(n:int)->bool` that given a positive integer `n` checks if it is prime.

16. Define a function `gcd(m:int,n:int)->int` that returns the greatest common divisor of `m` and `n` computed using Euclides' algorithm and iteration:

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

17. Define a function `lcm(m:int,n:int)->int` that returns the least common multiple of `m` and `n`.

3 Programming with loops and nested lists

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

1. Define an iterative function `print_lengths(m:List[list]])` that prints the length of each list in `m`.

2. Define an iterative function `max_length(m:List[list]])->int` that returns the length of the longest list in `m`.

3. Define an iterative function `total_length(m:List[list])->int` that returns the combined length of list in `m`.

4. Define an iterative function `sum_2d(m:List[List[int]])->int` that returns the sum of all elements of `m`.

[1]To write a type hint for an argument `https://docs.python.org/3/library/typing.html#typing.Callable`.

5. Define an iterative function `count_2d(m:List[list],v:Any)->int` that returns the number of occurrences of `v` in `m` (without creating intermediate lists).

6. Define an iterative function `max_2d(m:List[List[int]])->int` that returns the maximum element in `m` (without creating intermediate lists).

7. Define a function `increment_2d(m:List[List[int]])` that increments every number in `m` by one.

8. Define a function `parity_2d(m:List[List[int]])` that replaces each element in `m` by `0` if even and by `1` if odd.

9. Define a function `chunks(l:list,n:int)->List[list]` 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]]
```

10. Define a function `exact_chunks(l:list,n:int)->List[list]` 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]]
```

11. Define a function `dealing(l:list,n:int)->List[list]` 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]]
```

12. Define a function `exact_dealing(l:list,n:list)->List[list]` 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]]
```

13. Define a function `differences(l:List[int])->List[List[int]]` 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]]
```

14. Define a function `pascal(n:int)->List[List[int]]` 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]]`.

15. Write a function `trim_ends(m:List[list],w:int)` that trims every list in `m` to have at most `w` elements by deleting indexes at the end.

16. Write a function `fill_ends(m:List[list],d:Any)` that fills every list in `m` adding `d`s to its end such that they all have the same length.

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

4 Programming with loops and matrices

In this section you will work with matrices[2] A possible representation of a matrix in Python is as a list of lists where each of the inner list represents a row. For instance, the matrix below on the left is represented as the list below on the right.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
[[1, 2, 3],
 [4, 5, 6]]
```

Not all values of type `List[List[float]]` represent matrices. To communicate in our code when a value is (or is expected to be) a matrix, we introduce a distinct subtype of `List[List[float]]` using `typing.NewType`[3].

```
from typing import NewType
Matrix = NewType('Matrix',List[List[float]])
```

Now we can use `Matrix` in type hints and as a type constructor (e.g., `Matrix([[1,2]])` tells tools and programmers to regard `[[1,2]]` as a value of type `Matrix`) however, the default behaviour of this constructor is too lax for our aims: it does not enforce the additional requirements on dimensions (how could `NewType` know about them?). To enforce these constraints we need to define a new function.

```
def matrix(m:List[List[float]])->Matrix:
  """Returns m as a value of type Matrix.
  Raises: ValueError if m does not represent a matrix."""
  if is_matrix(m): # TODO: exercise 1
    return Matrix(m)
  else:
    raise ValueError('The argument does not represent a matrix.')
```

Finally, we define a function for retrieving the dimensions of a matrix.

```
def dimensions(m:Matrix)->[int]:
  """Returns a list with the number of rows and columns of the
  matrix m."""
  # from the function contract we know that m is a matrix so it is safe
  # to assume that it has at least one row and column.
  return [len(m),len(m[0])]
```

Observe that because `NewType` defines `Matrix` as a subtype of `List[List[float]]`, programmers and code have complete access to list operations and not all of them make sense on matrices—later in the course we will see how to avoid this "leak" and define a type for matrices that hides such details.

[2] In this context, a matrix is a rectangular array (a table) of numbers, called *entries*. The number or rows and columns of a matrix are its *dimensions*; a $m \times n$ matrix is a matrix with m rows and n columns. We ignore degenerate cases or matrices with 0 rows or 0 columns.

[3] `https://docs.python.org/3/library/typing.html#newtype`

1. Define an iterative function `is_matrix(m:List[List[float]])->bool` that checks whether `m` represents a matrix.

2. Define a function `print_matrix(m:Matrix)->None` 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
```

3. Define a function `print_table(m:Matrix)->None` 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.)

4. Define a function `zeros(r:int,c:cols)->Matrix` that returns a matrix with `r` rows and `c` columns whose entries are all zeros.

5. Define a function `identity(n:int)->Matrix` 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
```

6. Define functions `del_col(m:Matrix,j:int)` and `del_row(m:Matrix,i:int)` that delete the `j`-th column and `i`-th row from the given matrix `m`.

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

7. Define a function `scalar_sum(m:Matrix,s:float)` that takes a matrix `m` and a number `s` and increments each element of `m` by `s`.

8. Define a function `scalar_prod(m:Matrix,s:float)` that takes a matrix `m` and a number `s` and multiplies each element of `m` by `s`.

9. Define a function `matrix_prod(m1:Matrix,m2:Matrix)->Matrix` that returns the product of `m1` and `m2` (or raises a suitable error if their dimensions are incompatible).

10. Define a function `transposed(m:Matrix)->Matrix` that takes a matrix `m` and returns its transposed matrix.

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

11. Define `SquareMatrix` as a subtype of `Matrix` (and the relevant helper functions) for representing matrices that have the same number for rows and columns. Define a function `transpose(m:SquareMatrix)->None` that takes a square matrix `m` flips it over the diagonal (without creating a new matrix).

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