POETRY OF PROGRAMMING

CLOJURE CODING PROBLEMS
V2018.06.11

People often use programming and coding as synonyms. It is probably better to distinguish between the two terms, since coding is just one activity in the process. Solving a programming task has two conceptual steps:

(1) imagining a computational process that produces the desired result or behaviour, and
(2) the coding part, i.e. describing the envisioned process in a particular language.

These exercises are only for practising coding. This is the easier part. The idea of the solution is already described here in more or less plain English. The idea for the solution is language independent (barring some edge cases). These exercises are for developing your skill of expressing ideas in Clojure.

(1) **Distinct numbers.** Write a function `distinct-numbers`, that takes a collection of arbitrary data items and returns a collection of all numbers contained in the input, without repetitions.

```
(distinct-numbers [\a 1 "two" 5 :a 2 1])
(1 5 2)
(distinct-numbers ["Hello" "world" "!"])
()
(distinct-numbers [0 0 0 0 0 0 0 0])
(0)
```

Plan: We filter the input collection by a built-in predicate `number?`, deciding whether a given data item is a number or not. In order to avoid duplicates, we turn the result of filtering into a set. As the test cases suggest, we can convert that back to a sequence. Also, producing a sequence of distinct elements is a common thing to do, there should be a built-in function for that, it's worth checking.

(2) **Sum of negatives, sum of positives.** Write a function `sums`, which accepts a collection of numbers and sums the negative and positive numbers separately. It returns a vector containing the negative sum and the positive sum, in this order.

```
(sums [-1 2 0 -3 5])
[-4 7]
(sums [])
[0 0]
```

Plan: Separating the numbers by filtering with existing predicate functions `neg?` and `pos?`. For increased readability, these subsequences, or their sums my be temporarily bound to symbols in a `let` statement.

(3) **Database of squares.** Write a function `squares`, that takes a collection of numbers and returns a hash-map associating its square to each number.

```
(squares [0 1 11 2])
{0 0, 1 1, 11 121, 2 4}
```

Plan: We can simply `zipmap` the input collection with its transformation by using a (possibly anonymous) squaring function.

(4) **Integer numbers from numerical characters.** Write a function that accepts sequences of characters. These are all digits of decimal numbers. The function should return the corresponding integer.

```
(chars->int [\1 \2 \3])
123
(chars->int [\7 \2])
72
(chars->int [])
0
```

Plan: The most sensible solution is to find a built-in function, which can turn meaningful strings into other data types based on their content. Indeed, `read-string` is such a universal function. As a preparation, all we need to do is to turn the sequence of characters into a string.

In case we want a more challenging way, we can do the conversion ourselves. This is a good example of a reducing algorithm. The result so far is the number obtained by processing the first j digits. The next item is the next digit in the collection, so we multiply the result so far by 10 and the next digit to it. We also have to convert characters to their single digit values. The traditional way is to use their ASCII/Unicode values. If this sounds mysterious, then one can just build a little hash-map, in which keys are numerical characters and the values are the corresponding integer values.

(5) **Bank account balance.** Write a function `balance` that take a list of transactions and determines the balance of the bank account. The transactions are given in a vector. A single transaction is represented by a keyword `:W` or `:D` representing withdrawal and deposit followed by the transaction amount.

```
(balance [:D 100 :W 50 :W 49])
1
(balance [])
0
(balance [:W 11])
-11
```

Plan: First we need to separate individual transactions, representing each with a 2-element sequence. This can be done by the built-in `partition` function. Once we have a sequence of these transactions, we can filter them by the transaction type and sum them up accordingly.

(6) **Identity matrix.** Write a function that returns the identity matrix of size n, represented as a vector of vectors.

```
(identity-matrix 2)
[[1 0] [0 1]]
(identity-matrix 3)
[[1 0 0] [0 1 0] [0 0 1]]
(identity-matrix 4)
[[1 0 0 0] [0 1 0 0] [0 0 1 0] [0 0 0 1]]
```

Plan: There is no need to worry about what the identity matrix is. The pattern shown by the test cases is simple: the ith vector is a vector of zeroes, except a single 1 in the ith position. We need to produce these n vectors, so this activity fits into the scheme of functional collection processing. We turn numbers into vectors. By what function? We need to define a function `zeroes-with-one`. It creates a zero vector of the given size with a one at the given position. So this function has two inputs.

```
(zeroes-with-one 4 1)
[0 1 0 0]
(zeroes-with-one 6 0)
[1 0 0 0 0 0]
(zeroes-with-one 6 5)
[0 0 0 0 0 1]
```

How to write this? `[0 0 1 0 0 0]` can be thought of as a concatenation of sequences, 2 zeroes, a 1, and 3 zeroes. Sequences of zeroes can be produced by the `repeat` built-in function. Once we have this function, we can just map this over the first n numbers. Well, we still need to wrap it into an anonymous function to pre-fill the size argument.