Excersise 2

Contents

Prelude

Instead of Python this solution uses Hy to solve the assignments. Hy is a Lisp dialect implemented in Python which can use the Python ecosystem, including the standard library and third party modules. It has many functions, macros, and special forms to provide Python's language features in a ”lispy” syntax.

Let us start with some imports of macros and functions used. The `factorial` function is used to check the results of our own implementations in assignment 5 b.

```
(require [hy.contrib.loop [loop]])
(import [math [factorial sqrt]])
```

Assignment 1

Q Read an integer from the command line (hint: `input` function). Double the number and if it is positive. Halve the number if it is negative. Print the result.

A Asking the user for integer numbers is part of several of the assignments, this subproblem is solved its own function which repeatedly asks the user until a suitable input was made.

```
(defn ask-for-integer []
  (while True
    (try (setv result (int (input "Enter integer: ")))
      (except [ValueError] (print "Not a number, please try again."))
      (else (break))))
  result)
```

The DRY Principle

Functions are one way to follow the DRY principle: „**D**on't **R**epeat **Y**ourself“. Programmers strive to avoid redundancy in code and data.

Testrun:

```
=> (ask-for-integer)
Enter integer:
Not a number, please try again.
Enter integer: x
Not a number, please try again.
Enter integer: 42
42
```

The solution asks the user for a number with the `ask-for-integer` function just defined above and then prints the result of either dividing or multiplying the value by two.

```
(defn assignment-1 []
  (let [value (ask-for-integer)]
    (print ((if (neg? value) / *) value 2))))
```

Testrun:

```
=> (assignment-1)
Enter integer: 5
10
=> (assignment-1)
Enter integer: -5
-2.5
```

Assignment 2

Q Define a list of at least ten strings. Print every second string.

A This seems straight forward. The function declares a list with the first ten letters of the alphabet, "a" to "j", and then uses `take-nth` to get every second element to `print` it.

```
(defn assignment-2 []
  (let [strings ["a" "b" "c" "d" "e" "f" "g" "h" "i" "j"]]
    (for [s (take-nth 2 strings)] (print s))))
```

Testrun:

```
=> (assignment-2)
a
c
e
g
i
```

Assignment 3

Q Read numbers from the command line. Add them up until 0 is entered. Afterwards print the sum.

A As reading a sequence of numbers from the command line, with zero as end marker, is part of both this and assignment 4 we write a short function for this. DRY again.

First we inform the user what we expect and how the input can be ended. Then the function uses `iter` to create an iterable which repeatedly calls `ask-for-integer` defined in assignment 1 until that function returns a zero.

```
(defn ask-for-integers []
  (print "Enter integers. Enter 0 to end.")
  (iter ask-for-integer 0))
```

Testrun:

```
=> (list (ask-for-integers))
Enter integers. Enter 0 to end.
Enter integer: 1
Enter integer: -2
Enter integer: x
Not a number, please try again.
Enter integer: 3
Enter integer: 0
[1, -2, 3]
```

Now the actual function becomes trivial: Just sum the elements of the iterable returned by `ask-for-integers`.

```
(defn assignment-3 []
  (print (sum (ask-for-integers))))
```

Testrun:

```
=> (assignment-3)
Enter integers. Enter 0 to end.
Enter integer: 1
Enter integer: 2
Enter integer: 3
Enter integer: -4
Enter integer: 0
2
```

Assignment 4

Q Read numbers from the command line. Sum them up if they are positive, ignore them if they are negative and stop when 0 is entered. Use `continue` and `break`!

A The function starts by setting the local variable `result` to 0. Then it goes with a `for` loop over the numbers entered by the user based by repeatedly calling `ask-for-integer` defined in assignment 1.

For each `value` it then checks two conditions from the question and acts accordingly. If the value is 0 the `for` loop is left by `break`. If the value is negative the loop `continues` with the next value from the user. That leaves only positive values so the result is updated by adding the value to the current result at the

end, if the loop was not left by `break` and the last expression was not skipped by `continue`.

```
(defn assignment-4 []
  (setv result 0)
  (for [value (repeatedly ask-for-integer)]
    (if (zero? value) (break))
    (if (neg? value) (continue))
    (setv result (+ result value)))
  (print result))
```

Testrun:

```
=> (assignment-4)
Enter integer: 1
Enter integer: -2
Enter integer: 3
Enter integer: -4
Enter integer: 0
4
```

The `if`s are ordered in a way there are no unneccessary checks made. First we check for zero because then the other two checks are irrelevant. Then the negativ check is made because if it is true we can skip the positive check.

But wait, didn't we define `ask-for-integers` in assignment 3 for repeatedly asking the user for numbers? Yes we did. But if we used it here we would have a hard time to justify the zero test and the `break` because `ask-for-integers` already stops at zero, so the first condition is never met and we have no use for `break`.

`continue` is also only neccessary because of the way the tests are odered and expressed. Instead of skipping the addition at the end of the `for` loop, we could reverse the test and only execute the addition if the result is positive. All this leads to a much simpler and shorter function:

```
(defn assignment-4-simplified []
  (setv result 0)
  (for [value (ask-for-integers)]
    (if (pos? value) (setv result (+ result value))))
  (print result))
```

Testrun:

```
=> (assignment-4-simplified)
Enter integers. Enter 0 to end.
Enter integer: 1
Enter integer: -2
Enter integer: 3
Enter integer: -4
Enter integer: 0
4
```

A function which still is not the most elegant and concise way to solve this problem. We want to `filter` the numbers so only positive ones are summed. Extending the solution from assignment 3 we get:

```
(defn assignment-4-concise []
  (print (sum (filter pos? (ask-for-integers)))))
```

Testrun:

```
=> (assignment-4-concise)
Enter integers. Enter 0 to end.
Enter integer: 1
Enter integer: -2
Enter integer: 3
Enter integer: -4
Enter integer: 0
4
```

Assignment 5

Q Try do solve at least one of assignment 5 b and assignment 5 c, if you can (yes, they are a bit tougher!)

Assignment 5 a

Q Write a function which returns the absolute difference between two numbers *a* and *b*: | *a*-*b* |. Call the function for five combinations for *a* and *b*.

A Given there is an `abs` function and the substraction is a very basic operation this almost seems to be too simple.

```
(defn assignment-5-a [a b]
  (abs (- a b)))
```

And a small function to test five pairs of numbers against their expected results.

```
(defn test-assignment-5-a []
  (for [[a b expected] [[1 2 1] [2 1 1] [2 2 0] [-1 1 2] [1 -1 2]]]
    (assert (= (assignment-5-a a b) expected))))
```

Testrun:

```
=> (test-assignment-5-a)
```

Assignment 5 b

Q Write a function that computes and returns the factorial of any positive integer. Use either a `for` loop, a `while` loop, or recursion. Call the function for 13 and print the returned factorial to the console.

A We don't just solve assignment 5 b *and* 5 c but also all three alternatives for looping: `for`, `while`, and recursion. Plus an implementation with no explicit code for looping at all, leveraging the power of the rich set of functional building blocks available in Python and therefore also in Hy.

As factorial is defined for positive numbers only we start with a small function to check if a number is negative. It throws an `ValueError` exception if the number is negative and returns the number otherwise. The check is in an own function to avoid repeating the code in each of the following implementations.

```
(defn check-positive [n]
  (if (neg? n) (raise (ValueError "value must be positive")) n))
```

Testrun:

```
=> (check-positive 42)
42
=> (check-positive -23)
Traceback (most recent call last):
  ...
ValueError: value must be positive
=> (check-positive 0)
0
```

factorial with for

After initialising the result to 1 the `for` loop iterates over the range of integer numbers from 1 to the given number. The upper bound is checked if it is positive and the increased by 1 because `range` yields the range of numbers *excluding* the given endpoint. Within the loop all numbers are multiplied to the result.

```
(defn assignment-5-b-1 [n]
  (setv result 1)
  (for [x (range 1 (inc (check-positive n)))]
    (setv result (* result x)))
  result)
```

factorial with while

First the function checks if the given number is positive. Then the result is initialised with 1. As long as the number is greater that one, the result is multiplied with the current number, which is decreased by one before the next iteration of the loop starts.

This is the most ”primitive”, imperative variant of the four alternatives presented here, as we had to spell out each step explicitly, in the exact order it is executed by the computer. Although the recursive factorial in the very next section has the same amount of expressions, the order of execution of each single one is not that obviously encoded, as initialising and updating the two variables looks more ”parallel” there.

```
(defn assignment-5-b-2 [n]
  (check-positive n)
  (setv result 1)
  (while (> n 1)
    (setv result (* result n)
          n (dec n)))
  result)
```

Recursive factorial

For recursion we use the `loop` and `recur` macros. `loop` is used to define and call an anonymous function and `recur` calls this function recursively. The advantage of `loop` and `recur` over defining an ordinary function is that those macros do a little extra work to do tail call optimization (TCO) for which the function must be tail recursive.

The recursive function gets the number to calculatue the factorial of and the result which is given as 1 at the first call. If the number is smaller than 1 the result is returned, else the function is called recursively with the number decreased by one and the accumulated result mulipliplied by the (non-decreased)

number.

```
(defn assignment-5-b-3 [n]
  (loop [[i (check-positive n)] [result 1]]
    (if (< i 1)
      result
      (recur (dec i) (* result i)))))
```

Without `loop` and `recur` it would look like this and there would be no TCO:

```
(defn assignment-5-b-3-without-loop [n]
  (let [f (fn [i result]
            (if (< i 1)
              result
              (f (dec i) (* result i))))]
    (f (check-positive n) 1)))
```

`factorial` with functional building blocks

Practising different kind of looping techniques and learning when to use them is important. So is knowing the functional building blocks to write concise and elegant solutions without explicit loops. Here we can use `reduce`. It takes a binary function, an iterable of values, and an initial value, and applies the given function to each of the values from the iterable, using the call from the last result as first argument — or the given initial argument for the very first call.

```
(defn assignment-5-b-4 [n]
  (reduce * (range 1 (inc (check-positive n))) 1))
```

Testing `factorial`

For testing all the factorial implementations we first calculate the expected result by means of the `factorial` function from the `math` module. Then all four variations defined above will be called an their result is `asserted` to be the expected.

```
(defn test-assignment-5-b []
  (let [n 13
        expected (factorial n)]
    (for [f [assignment-5-b-1
             assignment-5-b-2
             assignment-5-b-3
             assignment-5-b-4]]
      (let [result (f n)]
        (assert (= result expected))
        (print result)))))
```

Testrun:

```
=> (test-assignment-5-b)
6227020800
6227020800
6227020800
6227020800
```

Assignment 5 c

Q Write a function that decomposes any positive integer into its prime factors and returns them as a list. Call the function for 12341234 and print the list of prime factors to the console.

A The basic idea for a solution is to to test numbers from 2 on upwards as candidates for (prime) factors and each time a candidate divides the number without a remainder to take the quotient as new number and note down the candidate as a prime factor. Although it seems tempting to determine prime numbers upfront to test them as factors, that step would not be necessary and will not speed up the solution unless we would cache those prime numbers for multiple calls to the function.

The implementation uses two simple optimizations:

- We do not test every number from 2 on upwards but just 2 and then every *odd* number greater than 2;
- and we do not test the numbers up to the current number we want to factor but just until the candidate reaches the square root of that number, as between that and the number there can not be another factor.

First we create an iterable with the candidates by chaining the list containing the 2 and an iterable counting from 3 on in steps of 2. In other words odd numbers.

Then we start a loop function with the number, its square root as upper limit for the candidates, and an empty list for the result as arguments.

After getting the next candidate it is checked against the current limit value. If the limit is lower than the candidate we are done and know the current number is itself the last prime factor needing to be `cons`ed to the result and being the result of the function.

Otherwise the quotient and the remainder for the number and the candidate are calculated. If the remainder is zero the loop function is called recursively with the quotient as the new number to factor, its square root as new limit and a result that is extended by the candidate. For a non-zero remainder the loop function is called with the same arguments from the previous call to check the next candidate.

```
(defn assignment-5-c [n]
  (let [candidates (chain [2] (count 3 2))]
    (loop [[number n] [limit (sqrt n)] [result []]]
      (let [candidate (next candidates)]
        (if (< limit candidate)
          (cons number result)
          (let [[quotient remainder] (divmod number candidate)]
            (if (zero? remainder)
              (recur quotient (sqrt quotient) (cons candidate result))
              (recur number limit result)))))))))
```

The result to test against was calculated with the linux command line tool `factor`. As `cons` prepends elements to the list, the comparison value goes from the highest to the lowest prime factor.

```
(defn test-assignment-5-c []
  (let [result (assignment-5-c 12341234)]
    (assert (= result [617 137 73 2]))
    (print result)))
```

Testrun:

```
=> (test-assignment-5-c)
[617, 137, 73, 2]
```