Recursion, Generators and Exceptions

Solve the following exercises and upload your solutions to Moodle until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. Unless explicitly stated otherwise, you can assume correct user input and correct arguments.

Exercise 1 – Submission: `ex1.py` 20 Points

Write a function `fib(n: int) -> int` that calculates and returns the `n`-th Fibonacci number. The Fibonacci sequence is defined as follows:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

If `n` is negative, -1 must be returned. You can assume correct arguments (no incorrect data types). In contrast to the last assignment, use recursion to solve this exercise, i.e., loops are *not allowed*.

Exercise 2 – Submission: `ex2.py` 20 Points

Write a generator function `gen_range(start: int, stop: int, step: int = 1)` that yields values similar to those produced by the built-in `range` (which you are *not allowed* to use). Given a starting integer `start`, continuously yield integers with a step size of `step` until `stop` (exclusive) has been reached. In addition, your function should do the following:

- If any of `start`, `stop` or `step` are not integers, raise a `TypeError`.
- If `step` is an integer with the value 0, raise a `ValueError`.

Example function calls and results:

```
list(gen_range(0, 10)) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list(gen_range(0, 10, 3)) = [0, 3, 6, 9]
list(gen_range(0, 10, -1)) = []
list(gen_range(10, 0)) = []
list(gen_range(10, 0, -2)) = [10, 8, 6, 4, 2]
list(gen_range(-10, -3, 2)) = [-10, -8, -6, -4]
list(gen_range(0.0, 10)) -> TypeError
list(gen_range(0, 10, 0)) -> ValueError
```

Hints:

- Create appropriate and useful error/exception messages.
- You can check if some object is of a certain data type with `isinstance(OBJECT, TYPE)`, e.g., `isinstance(start, int)` to check if `start` is an integer object.

Exercise 3 – Submission: ex3.py 25 Points

Write a function `binary_search(elements: list, x) -> bool` that applies a binary search on the list `elements` while searching for the value `x`. The binary search algorithm works as follows:

- Given a sequence sorted in ascending order (you can assume that `elements` is a list that is already correctly sorted), the middle element is compared to `x`.
- If `x` is equal to the middle element, `True` is returned (`elements` contains `x`).
- If `x` is smaller than the middle element, the left half of `elements` is checked, again by selecting the middle element.
- If `x` is bigger than the middle element, the right half of `elements` is checked, again by selecting the middle element.
- These steps are repeated until either the value is found or a half is empty, in which case `False` is returned (`elements` does not contain `x`).

Both the content of `elements` and `x` can be arbitrary objects. This means that the comparison operations with `<` (smaller) and `>` (bigger) might fail with a `TypeError`. If so, your function must catch this error and return `False`. Implement this function in a recursive manner. You are *not allowed* to use loops or the list-contains check `x in elements`.

Example function calls and results:

```
my_sorted_list = [1, 2, 5, 7, 8, 10, 20, 30, 41, 100]
binary_search(my_sorted_list, 1) -> True
binary_search(my_sorted_list, 20) -> True
binary_search(my_sorted_list, 21) -> False
binary_search(my_sorted_list, "hello") -> False
```

Hints:

- In case `elements` (or any split off half thereof) has an even number of elements, you can choose the middle element by either rounding up or down the index, e.g., if there are 10 elements with indices from 0 to 9, the middle element index would be 4.5, for which you can choose either 4 or 5 as index (both are equally fine).

Exercise 4 – Submission: ex4.py 15 Points

Write a function `flatten(nested: list) -> list` that flattens an arbitrarily nested list (you can assume correct arguments). Use recursion to implement this function.

Example function calls and results:

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

Hints:

- To check if an object is a list, you can use `isinstance(my_object, list)`.

Exercise 5 – Submission: ex5.txt 20 Points

Consider the following code with custom exceptions ErrorA, ErrorB and ErrorC (they are all independent, i.e., none of them is a special case of another one):

```
def f(x: int):
    try:
        g(x)
        print("f1")
    except ErrorA:
        print("f2")
        raise ErrorC
    except ErrorB:
        print("f3")
    else:
        print("f4")
    print("f5")

def g(x: int):
    try:
        h(x)
        print("g1")
    except ErrorA:
        print("g2")
        if x < -10:
            raise ErrorC
        print("g3")
    finally:
        print("g4")

def h(x: int):
    try:
        if x < 0:
            raise ErrorA
        if x > 10:
            raise ErrorB
    finally:
        print("h1")
    print("h2")
```

Determine the output of the function f with the following four arguments without actually running the code (the goal is to understand the program flow): f(1), f(-1), f(15), f(-15). Write your answers to the text file ex5.txt in the following format (one line per answer):

```
f(ARG) -> X1 X2 ... Xn
```

where ARG is one of the four input arguments from above and Xi are either space-separated print outputs or the error in case the function call ends with an error. Here is an example file content (the examples are incorrect, they are just for demonstrating purposes!):

```
f(1) -> f1 f2 g1 h1
f(-1) -> f3 h2 ErrorB
f(15) -> h1 h2 f1 f5 g2
f(-15) -> g1 h2 f3 ErrorA
```