Functions

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). Use a loop to solve this exercise.

Example function calls and results:

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(7) = 13
fib(-2) = -1
```

Exercise 2 – Submission: `ex2.py` 15 Points

Write a function `clip(*values, min_=0, max_=1) -> list` that returns a list of clipped values based on arbitrary many input values `*values` (integers or floats), where clipping is defined as follows:

- If a value is smaller than `min_`, append `min_` to the list.
- If a value is bigger than `max_`, append `max_` to the list.
- Otherwise, append the value to the list.

If `*values` is empty, an empty list must be returned. You can assume correct arguments (no incorrect data types).

Example function calls and results:

```
clip() = []
clip(1, 2, 0.1, 0) = [1, 1, 0.1, 0]
clip(-1, 0.5) = [0, 0.5]
clip(-1, 0.5, min_=-2) = [-1, 0.5]
clip(-1, 0.5, max_=0.3) = [0, 0.3]
clip(-1, 0.5, min_=2, max_=3) = [2, 2]
```

Exercise 3 – Submission: `ex3.py` 15 Points

Write a function `create_train_test_splits(data: list, train_size: float) -> tuple` that creates two data splits (train and test) as follows:

- The parameter `data` specifies a list of arbitrary elements, which is considered the entire data set from which the following two splits (training and test) are created. The splits are also lists.
- The parameter `train_size` specifies the percentage of the training split (you can assume correct arguments, i.e., floats in the range $(0, 1)$). This means that the first `train_size` percent of `data` should be part of the training split. In case the percentage does not lead to an integral split size, use the floor value, i.e., cut off the decimal part. Example: `len(data)` $= 10$ and `train_size` $= 0.67 \rightarrow$ `train_split_len` $= 6$ (from originally 6.7).
- The remaining percentage (and thus remaining elements of `data`) is attributed to the test split.
- Return the training split and test split as a tuple.

Example function calls and results:

```
create_train_test_splits([], 0.5) = ([], [])
create_train_test_splits(list(range(10)), 0.5) = ([0, 1, 2, 3, 4], [5, 6, 7, 8, 9])
create_train_test_splits(list(range(10)), 0.67) = ([0, 1, 2, 3, 4, 5], [6, 7, 8, 9])
```

Hints:

- It might happen that the percentage leads to an empty split (i.e., an empty list), which is fine.
- Make sure that the combined number of split elements is the same as the original input list, i.e., do not drop any elements.

Exercise 4 – Submission: `ex4.py` 25 Points

Write a function `round_(number, ndigits: int = None)` that rounds a given number (integer or float) to `ndigits` precision (you can assume correct arg data types). The function works as follows:

- If `ndigits` is `None`, the rounding is performed to 0 decimal places, and an integer is returned.
- In all other cases, a float is returned, and the rounding is performed to `ndigits` precision.
- If `ndigits` is negative, the rounding is performed on the integral part of the input number, e.g., for `number` $= 777$ and `ndigits` $= -1$, the result should be 780.
- You are *not allowed* to use the built-in function `round`.

Example function calls and results (results might differ slightly because of floating point arithmetic):

```
round_(777.777) = 778
round_(777.777, 0) = 778.0
round_(777.777, 1) = 777.8
round_(777.777, 2) = 777.78
round_(777.777, 3) = 777.777
round_(777.777, 4) = 777.777
round_(777.777, -1) = 780.0
round_(777.777, -2) = 800.0
round_(777.777, -3) = 1000.0
round_(777.777, -4) = 0.0
```

Hints:

- As mentioned above, the rounded results might not be exactly precise due to floating point arithmetic. For example, `round_(777.777, 1)` might result in `777.8` or `777.8000000000001` (or comparable numbers), depending on the implementation. Both results are fine.
- The module operator `%` might be useful to solve the task. For example, `some_float % 1` will return the fractional part of some floating point number, or `some_float % 0.1` the fractional part starting at the second decimal place. As mentioned above, the results obtained from the module operation might not be precise because how floating point arithmetic works, which is okay.

Exercise 5 – Submission: `ex5.py` 25 Points

Write a function `sort(elements: list, ascending: bool = True)` that sorts the specified list in-place, i.e., the list is changed directly. The function does not return anything. The parameter `ascending` controls whether the list should be sorted in ascending or in descending order. To implement the sorting functionality, take a look at the bubble sort algorithm (you can find a visual example here (ascending order)):

- Two consecutive elements at index positions i and $i + 1$ are compared to each other.
- If the first element is bigger/smaller (ascending/descending order) than the second element, they are swapped.
- Then, i is incremented, and the comparison and swapping is repeated.
- Afterwards, the biggest/smallest (ascending/descending order) element will be at the end of the list.
- All of this is then repeated except for this lastly processed element (since it is already in the correct position) until there are no more elements to compare.

You are *not allowed* to use the built-in function `sorted` or the list method `some_list.sort()`. You do not need to implement any optimizations (of course, you are free to do so if you want). However, you will need to swap list elements in two code locations: one time for the ascending case and one time for the descending case. To avoid code duplication, create an additional helper function that does exactly this, and then use this helper function. The helper function name and its parameters are completely up to you (you can even create a local/nested function).

Example function calls and results:

```
some_list = [1, 3, 0, 4, 5]
sort(some_list) -> some_list = [0, 1, 3, 4, 5]
sort(some_list, ascending=False) -> some_list = [5, 4, 3, 1, 0]
```