Classes: Advanced Topics

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. You are allowed to implement additional attributes and methods as long as the original interface remains unchanged.

Exercise 1 - Submission: ex1.py

40 Points

Create a class Complex that models complex numbers. The class has the following instance attributes:

• real: float

Represents the real part of the complex number.

• imaginary: float

Represent the imaginary part of the complex number.

The class has the following instance methods:

• __init__(self, real: float, imaginary: float)
Sets both instance attributes.

• __eq__(self, other)

If other is not an instance of Complex, NotImplemented is returned. If it is an instance, True is returned if both attributes real and imaginary of other are equal to the ones of self, False otherwise.

• __repr__(self)

Returns the following string format: "Complex(real=<real>, imaginary=<imag>)", where <real> is the real part of the complex number and <imag> is the imaginary part.

• __str__(self)

Returns the following string format: "<real> <sign> <imag>i", where <real> is the real part of the complex number, <imag> is the imaginary part (followed by the character i), and <sign> is either the plus character + if the imaginary part is positive or the minus character - if the imaginary part is negative. Example: "1.2 - 5.4i"

• __abs__(self)

Returns the absolute value of this Complex object. The absolute value of a complex number is defined as $|a+bi| = \sqrt{a^2 + b^2}$, where a and b are the real and imaginary parts, respectively.

• __add__(self, other)

If other is an instance of Complex, adds other to self and returns a new Complex object with the result of this addition. Otherwise, NotImplemented is returned.

• __iadd__(self, other)

If other is an instance of Complex, adds other to self in-place and returns self. Otherwise, NotImplemented is returned.

Moreover, add the following static method (@staticmethod):

• add_all(comp: "Complex", *comps: "Complex") -> "Complex"

Adds comp and all numbers in *comps together and returns a new Complex object containing this sum. None of the input arguments must be changed, i.e., all complex numbers specified by comp and *comps must remain the same.

Example program execution:

```
c1 = Complex(-1, -2)
c2 = Complex(2, 4)
c3 = Complex(1, 2)
print(c1 == c3, c1 + c2 == c3)
print(repr(c1))
print(c1)
print(abs(c1))
print(c1 + c2)
c1 += c3
print(c1)
print(Complex.add_all(c2, c2, c3))
    c1 + 1
except TypeError as e:
    print(f"{type(e).__name__}: {e}")
Example output:
False True
Complex(real=-1, imaginary=-2)
-1 - 2i
2.23606797749979
1 + 2i
0 + 0i
5 + 10i
TypeError: unsupported operand type(s) for +: 'Complex' and 'int'
```

Exercise 2 – Submission: ex2.py

30 Points

Create a class Reader that enables index-based binary/bytes file read access. The class has the following instance methods:

- __init__(self, path: str) The string path points to the file that should be read. If path does not specify a file, a ValueError must be raised. The file is then opened in binary/bytes read mode "rb" and remains open until the Reader.close method is called (see below)
- close(self)

Closes the file that was opened in __init__. After invoking this method, the current Reader does not work anymore, i.e., a new Reader object must be created if the user wishes to read data from the file again.

• __len__(self)

Returns the number of bytes in the opened file.

• __getitem__(self, key)

Enables index-based access to the bytes of the opened file. The method works as follows:

- If key is an integer, it represents the file index position where a single byte must be returned (a bytes object of size 1). If key is out of range, i.e., an invalid index, an IndexError must be raised. Negative values must also be supported.
- All other data types result in a TypeError.

You are *not allowed* to read the entire file content at once. Only the bytes specified via <code>__getitem__</code> should be actually read into memory and returned.

```
Example file content (ex2_data.txt):
this is some text file
with 2 lines and special char \mu
Example program execution:
r = Reader("ex2_data.txt")
print(r[0])
print(r[1])
print(r[-1])
try:
    r["hi"]
except TypeError as e:
    print(f"{type(e).__name__}: {e}")
try:
    r[100]
except IndexError as e:
    print(f"{type(e).__name__}: {e}")
Example output:
b't'
b'h'
b'\xb5'
TypeError: indexing expects 'int', not 'str'
IndexError: Reader index out of range
```

Hints:

- For a given file handle fh as obtained by fh = open(some_file, "rb"), you can navigate through the file content by setting the current file index position via fh.seek(offset, whence), where offset is the byte offset that is added to the position specified by whence (possible values for whence: 0 = os.SEEK_SET = start of file, 1 = os.SEEK_CUR = current file position, 2 = os.SEEK_END = end of file).
- You can get the size of a file (in bytes) in two ways: Either by calling os.path.getsize(path), or, for a given file handle fh as obtained by fh = open(some_file, "rb"), by calling the method fh.seek(0, os.SEEK_END).

Exercise 3 – Submission: ex3.py

30 Points

Create a class Aggregator that continuously aggregates/sums up arbitrary objects of a fixed data type via Python's callable interface. The class has the following instance methods (the instance attributes are completely up to you):

- __init__(self, agg_type: type, ignore_errors: bool = True)

 agg_type specifies the type/class of the objects that will be aggregated in the method __call__,
 thereby potentially raising errors depending on ignore_errors (see __call__ below). You can
 assume that this type supports aggregation/summing up via the + operator, i.e., you do not
 have to worry about users specifying a type which does not support this operation.
- __call__(self, *args)

This method enables Aggregator instances to be callable (like a function), i.e., code such as $my_agg(x, y, ...)$ will roughly be translated to Aggregator.__call__($my_agg, x, y, ...$), see the example program execution below. The method description is as follows:

Using the + operator, continuously aggregates/sums up the objects specified by *args and returns the current aggregation/sum. If ignore_errors (see __init__ above) is True, objects that are not instances of agg_type (see __init__ above) are simply ignored. Otherwise, a TypeError must be raised. If the method is called without any arguments, the current aggregation/sum must be returned without any additional actions. If no values have been aggregated/summed up yet, None is returned.

For simplicity reasons, you can assume that the agg_type passed to __init__ remains fixed and never changes once the Aggregator instance was created.

Example program execution:

```
int_agg = Aggregator(agg_type=int)
int_agg(1, 2, 3)
int_agg(4, "hi", 5.1)
print(int_agg())
str_agg = Aggregator(agg_type=str, ignore_errors=False)
print(str_agg("this", " ", "is a test"))
try:
    str_agg(1)
except TypeError as e:
    print(f"{type(e).__name__}}: {e}")

Example output:

10
this is a test
TypeError: aggregation only works on type 'str', not 'int'
```