Classes: Advanced Topics

Solve the following exercises and upload your solutions to Moodle until the specified due date.

Important Information!

Please try to *exactly match the output* given in the examples (naturally, the input can be different). We are running automated tests to aid in the correction and grading process, and deviations from the specified output lead to a significant organizational overhead, which we cannot handle in the majority of the cases due to the high number of submissions.

Make sure to use the *exact filenames* that are specified for each individual exercise.

Also, use the provided unit tests to check your scripts before submission (see the slides Handing in Assignments on Moodle). Feel free to copy the example text from the assignment sheet, and then change it according to the exercise task to match the output as best as possible.

In this assignment, it is of *particular importance* to wrap the printing that you see in the example outputs in `if __name__=='__main__':`, as your exercises essentially only consist of a function definition. Example - let's say the task is to write a function that doubles the float value that is passed:

```
def double(var: float) -> float:
    return var*2

if __name__ == '__main__':
    print(double(3.4))
```

Unless explicitly stated otherwise, you can assume correct user input and correct arguments.

You are *not allowed* to use any concepts and modules that have not yet been presented in the lecture.

You are allowed in this assignment to implement additional or multiplication attributes and methods as long as the original interface remains unchanged.

Exercise 1 – Submission: `a9_ex1.py` 30 Points

Create a class `Vector` that represents a mathematical vector in n-dimensional space.

The class has the following instance attribute:

- `components: tuple`
  Represents the components of the vector as a tuple of numbers.

The class has the following instance methods:

- `__init__(self, components: list[float])`
  Initializes the vector with the given components. All elements in `components` must be numbers (integers or floats). If not, a `TypeError` must be raised.
- `__repr__(self)`
  Returns the string `"Vector(<components>)"`, where `<components>` is the tuple of the vector's components.
- `__str__(self)`
  Returns the string representation of the vector in angle brackets, e.g., `"<1, 2, 3>"`.
- `__eq__(self, other)`
  If `other` is not an instance of `Vector`, `NotImplemented` must be returned. If it is an instance, `True` must be returned if the `components` of `other` are equal to those of `self`, `False` otherwise.
- `__add__(self, other)`
  If `other` is an instance of `Vector` and has the same length as `self`, returns a new `Vector` where each component is the sum of the corresponding components of `self` and `other`. Otherwise, `NotImplemented` must be returned.
- `__radd__(self, other)`
  Implements addition when `Vector` is on the right side of the `+` operator.
- `__sub__(self, other)`
  Similar to `__add__`, but performs component-wise subtraction.
- `__neg__(self)`
  Returns a new `Vector` where each component is the negation of the corresponding component in `self`.
- `__mul__(self, scalar)`
  If `scalar` is a number (`int` or `float`), returns a new `Vector` where each component is multiplied by `scalar`. Otherwise, `NotImplemented` must be returned.
- `__rmul__(self, scalar)`
  Implements multiplication when `Vector` is on the right side of the `*` operator.
- `__len__(self)`
  Returns the number of components in the vector.
- `__getitem__(self, index)`
  Returns the component at the given `index`. Supports negative indices. If `index` is out of range, raises an `IndexError`.
- `__iter__(self)`
  Returns an iterator over the components of the vector.

Example program execution:

```
v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
print(v1)
```

```
print(repr(v2))
print(v1 == v2)
v3 = v1 + v2
print(v3)
v4 = v1 - v2
print(v4)
v5 = v1 * 2
print(v5)
v6 = -v1
print(v6)
print(len(v1))
print(v1[0], v1[1], v1[2])
for component in v1:
    print(component)
```

Example output:

```
<1, 2, 3>
Vector((4, 5, 6))
False
<5, 7, 9>
<-3, -3, -3>
<2, 4, 6>
<-1, -2, -3>
3
1 2 3
1
2
3
```

Notes:

- The multiplication in this exercise only supports scalar values. You do not need to implement the cross product or dot product.

Exercise 2 – Submission: `a9_ex2.py` 40 Points

Create a class `Time` that represents time in hours, minutes, and seconds.

The class has the following instance attributes:

- `hours: int`
  Represents the hours component of the time (non-negative integer).
- `minutes: int`
  Represents the minutes component of the time (integer between 0 and 59).
- `seconds: int`
  Represents the seconds component of the time (integer between 0 and 59).

The class has the following instance methods:

- `__init__(self, hours: int, minutes: int, seconds: int)`
  Initializes the time object with the given hours, minutes, and seconds. If any of the values are out of their expected ranges (e.g., minutes or seconds not between 0 and 59, hours negative or more than 23), a `ValueError` must be raised.
- `to_seconds(self)`
  Returns the total number of seconds represented by the `Time` object as an integer (see notes below).
- `from_seconds(cls, total_seconds)`
  A class method that returns a `Time` object specified only by the total number of seconds (necessary for the `__add__()` and `__sub__()` methods). If `total_seconds` is smaller than 0 or greater than $23 * 3600 + 59 * 60 + 59 = 86399$ (meaning greater than 23:59:59), a `ValueError must be raised`.
- `__repr__(self)`
  Returns the string `"Time(hours=<hours>, minutes=<minutes>, seconds=<seconds>)"`, where `<hours>`, `<minutes>`, and `<seconds>` are the corresponding attributes.
- `__str__(self)`
  Returns the time in the format `"HH:MM:SS"`, with leading zeros if necessary (e.g., `"09:05:03"`).
- `__eq__(self, other)`
  If `other` is not an instance of `Time`, `NotImplemented` must be returned. If it is an instance, `True` must be returned if the `hours`, `minutes`, and `seconds` of `other` are equal to those of `self`, `False` otherwise.
- `__lt__(self, other)`
  Implements the less-than operator `<`. Returns `True` if `self` represents an earlier time than `other`. If `other` is not an instance of `Time`, `NotImplemented` must be returned.
- `__add__(self, other)`
  If `other` is an instance of `Time`, returns a new `Time` object representing the sum of `self` and `other`. The addition should correctly handle overflow of minutes and seconds into hours. If `other` is an integer representing seconds, adds those seconds to `self` and returns a new `Time` object. Otherwise, `NotImplemented` must be returned.
- `__radd__(self, other)`
  Implements addition when `Time` is on the right side of the `+` operator.
- `__sub__(self, other)`
  If `other` is an instance of `Time`, returns the total number of seconds between `self` and `other` as an integer. If `other` is an integer representing seconds, subtracts those seconds from `self` and

returns a new `Time` object. If the result is negative, a `ValueError` must be raised. Otherwise, `NotImplemented` must be returned.

- `__int__(self)`
  Returns the total number of seconds represented by the `Time` object as an integer.

Example program execution:

```
t1 = Time(1, 30, 15)
t2 = Time(2, 45, 50)
print(t1)
print(repr(t2))
print(t1 == t2)
print(t1 < t2)
t3 = t1 + t2
print(t3)
t4 = t1 + 5000  # Add 5000 seconds to t1
print(t4)
t5 = 3600 + t1  # Add 3600 seconds to t1
print(t5)
difference = t2 - t1  # Difference in seconds
print(difference)
t6 = t2 - 5000  # Subtract 5000 seconds from t2
print(t6)
try:
    t7 = Time(24, 0, 0)
except ValueError as e:
    print(f"{type(e).__name__}: {e}")
```

Example output:

```
01:30:15
Time(hours=2, minutes=45, seconds=50)
False
True
04:16:05
02:52:35
02:30:15
4535
01:22:30
ValueError: hours must be between 0 and 23
```

Notes:

- The class should handle overflow properly when adding times (e.g., if minutes or seconds exceed 59 - the resulting time cannot exceed 23:59:59). For this purpose implement a separate method `to_seconds(self)`, because you will need the total number of seconds of your time object at various points in your class definition (not just for `__int__(self)`).
- When subtracting two `Time` objects, the result is the total number of seconds between them.
- When subtracting seconds from a `Time` object, a `ValueError` must be raised if the result is negative.

Exercise 3 – Submission: `a9_ex3.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 `__getitem__` should be actually read into memory and returned.

Example file content (use the provided `a9_ex3_data.txt` to replicate this):

```
this is some text file
with 2 lines and special char µ
```

Example program execution:

```
r = Reader("a9_ex3_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)`.