

Classes

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

Important Information!

We are running automated tests to aid in the correction and grading process, and deviations from the expected outputs lead to a significant organizational overhead, which we cannot handle in the majority of the cases due to the high number of submissions.

1. Please try to *exactly match the output* given in the examples (naturally, the input can be different). Feel free to copy the output text from the assignment sheet, and then change it according to the exercise task.

For example, if the exercise has an output of

Number of cables: XYZ

(where XYZ is some user input), do not write

The number of cables: XYZ

(additional **The** and lowercase **n**) or

Number of cables:XYZ

(missing space after the colon).

2. Furthermore, please don't have any lines of code that will be automatically executed when importing your module (except for what is asked by the exercise) as this will break our automated tests. Always execute your module before submitting to verify this !

For example, if you have some code to test your program and reproduce the example outputs, either comment/remove these lines or move them to the `"if __name__ == \"main\":"` section.

Important Note: In this assignment, we don't allow any Python built-in module except math module to use only sin and cos.

Exercise 1 – Submission: a7_ex1.py

30 Points

Create a class `Radian` that converts degree angles. The class has the following instance attribute:

- `degree: float`

Represents the degree angle.

The class has the following instance methods:

- `__init__(self, degree: float)`

Sets this instance attribute.

- `rad(self) -> float`

Returns the radian angle of this `degree` angle. The conversion from radians to degrees is defined as $(a * (pi/180.0))$, where a is the angle in a degree, and pi is equal to 3.14.

- `print(self)`

Prints both angles to the console. The outputs should be formatted with 2 decimal places. Format: The degree is <deg> and the radian is <rad>, where <deg> is the angle in a degree, <rad> is the angle in a radian.

Example program execution:

```
c = Radian(90)
print(c.rad())
c.print()
```

Example output:

```
1.57
The degree is 90.00 and the radian is 1.57
```

Exercise 2 – Submission: a7_ex2.py**20 Points**

This exercises is a bonus exercise !

Create a class `Rotate` that rotates the matrix. The class has the following instance attributes:

- `matrix: list[list]`

Represents the intended matrix to be rotated.

- `degree: float`

Represents the rotation angle in a degree .

- `inplace: bool`

Represents the change of the matrix in-place or not. If `inplace` is `True`, the replacement of the numbers should be done in place. Otherwise, the replacement should be done in a new matrix. The default `boolean` is `False`.

The class has the following instance methods:

- `__init__(self, matrix: list, degree: float, inplace=False)`

Sets these instance attributes.

- `rotation(self) -> list[list] | None`

Returns the rotated matrix at designated `degree` angle. For conversion from degree to radian, you must use the above class by importing it as `from a7_ex1 import Radian`. You can only use `sin` and `cos` from `math`, which is a Python built-in module. You can find an example in `matrix_rotation.txt` and should be done as follows:

- Find the coordinates of the matrix center.
- For each matrix element coordinate their relative position to the matrix center.
- To get the new relative positions, apply the `rotation matrix` for each relative position.

You can assume that the dimension of the matrix is `nxn` which `n` being an odd numbers, for example, `3X3`, `5X5`, `7X7`, `9X9`, etc. Moreover, you can assume the rotation degree will be either `90`, `180`, or `-90`. Please write your code based on those assumptions only.

Example program execution:

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
c1 = Rotate(matrix,90)
c11= c1.rotation()
print(c11)
print(matrix)
```

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
c2 = Rotate(matrix,-90, True)
c22= c2.rotation()
print(c22)
print(matrix)
```

```
matrix = [[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20],[21,22,23,24,25]]
```

```
c3 = Rotate(matrix,180)
c33 = c3.rotation()
print(c33)
```

Example output:

$[[3, 6, 9], [2, 5, 8], [1, 4, 7]]$
which corresponds to $c11 = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{bmatrix}$

$[[1, 2, 3], [4, 5, 6], [7, 8, 9]]$
which corresponds to matrix $= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

None
 which corresponds to $c22 = \text{None}$

$[[7, 4, 1], [8, 5, 2], [9, 6, 3]]$
which corresponds to matrix $= \begin{bmatrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{bmatrix}$

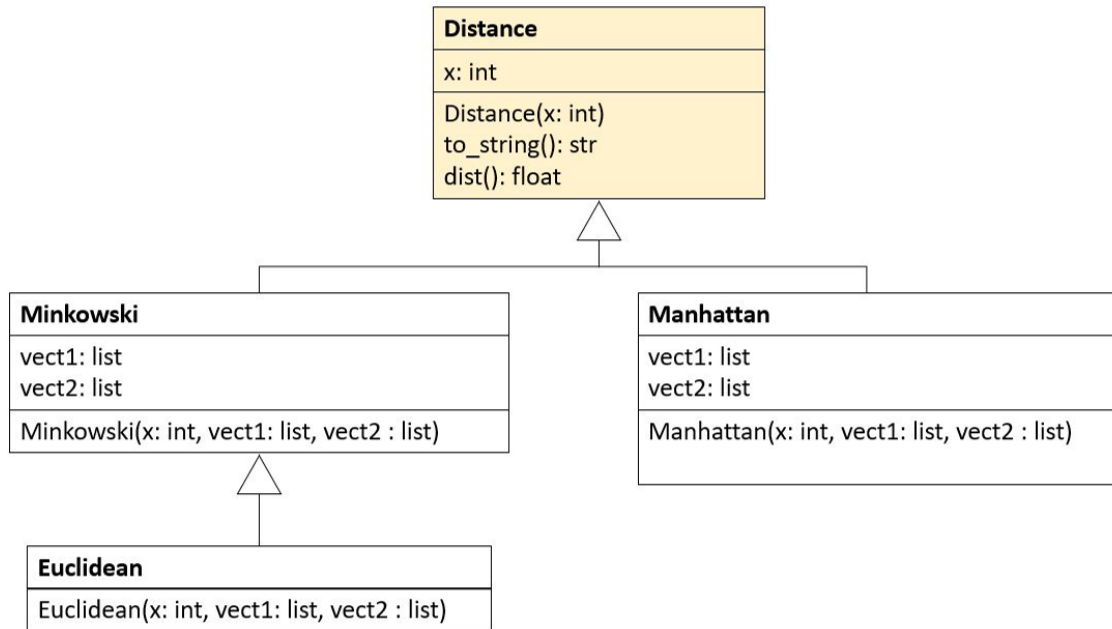
$[[25, 24, 23, 22, 21], [20, 19, 18, 17, 16], [15, 14, 13, 12, 11], [10, 9, 8, 7, 6], [5, 4, 3, 2, 1]]$
which corresponds to $c33 = \begin{bmatrix} 25 & 24 & 23 & 22 & 21 \\ 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$

Hints:

- To rotate the matrix please refer to this link [wikipedia](#).
- You can find an example of how to get the new coordinates in `matrix_rotation.txt`.
- You need to convert the angle from degrees to radians.

Exercise 3 – Submission: a7_ex3.py**25 Points**

You are given the following class hierarchy that models distances between 2 vectors. You have to implement the classes in this and the following exercises¹.



In this exercise, you have to implement the class `Distance`, which represents the base class of 1D vectors. The class has the following instance attributes:

- `x: int`

The number of vectors.

The class has the following instance methods:

- `__init__(self, x: int)`

Set instance attribute.

- `to_string(self) -> str`

Returns a string representation of the form `"Distance: the number of vectors =<x_value>"`, where `<x_value>` represents the value of the corresponding attribute.

- `dist(self) -> float`

Returns the distance between 2 vectors as a float, which must be implemented by all concrete subclasses. In the `Distance` class, a `NotImplementedError` is raised.

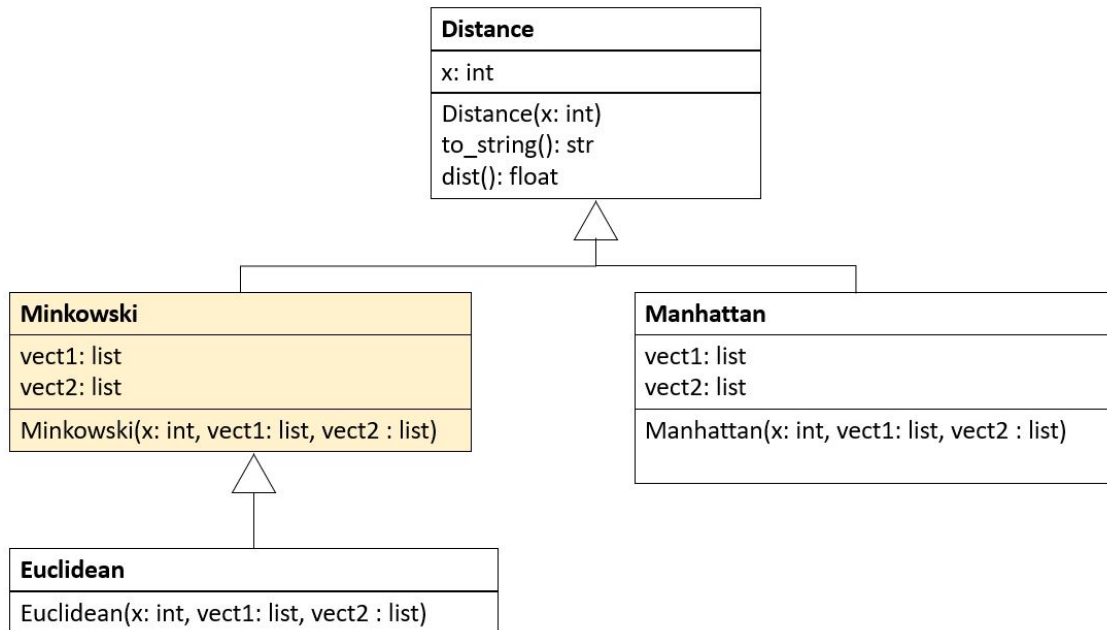
Hints:

¹The class hierarchy does not reflect the conceptual relationship 1:1

- In the `to_string` method, you need the name of the class. While this could be hard-coded, you could also use `type(x).__name__` to get the name of the type/class of some object `x`. This has the benefit that instances of any subclasses will also return their correct names. Alternatively, you can write a helper method that returns the name, and you override it in the subclasses.

Exercise 4 – Submission: a7_ex4.py**15 Points**

You are given the same class hierarchy as in the previous exercise that models 1D vectors.



In this exercise, you have to implement the concrete subclass `Minkowski`, which represents a Minkowski distance and extends the base class `Distance`. The class has the following additional instance attributes:

- `vect1: list`
The first vector.
- `vect2: list`
The second vector.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- `__init__(self, x: int, vect1: list, vect2: list)`
Sets both instance attributes (in addition to the attributes of the base class `Distance`).
- `to_string(self) -> str`
Returns a string representation of the form `"Minkowski: the number of vectors =<x_value>, vector_1=<vect1_value>, vector_2=<vect2_value>"`, where `<?_value>` represents the value of the corresponding attribute.
- `dist(self) -> float`

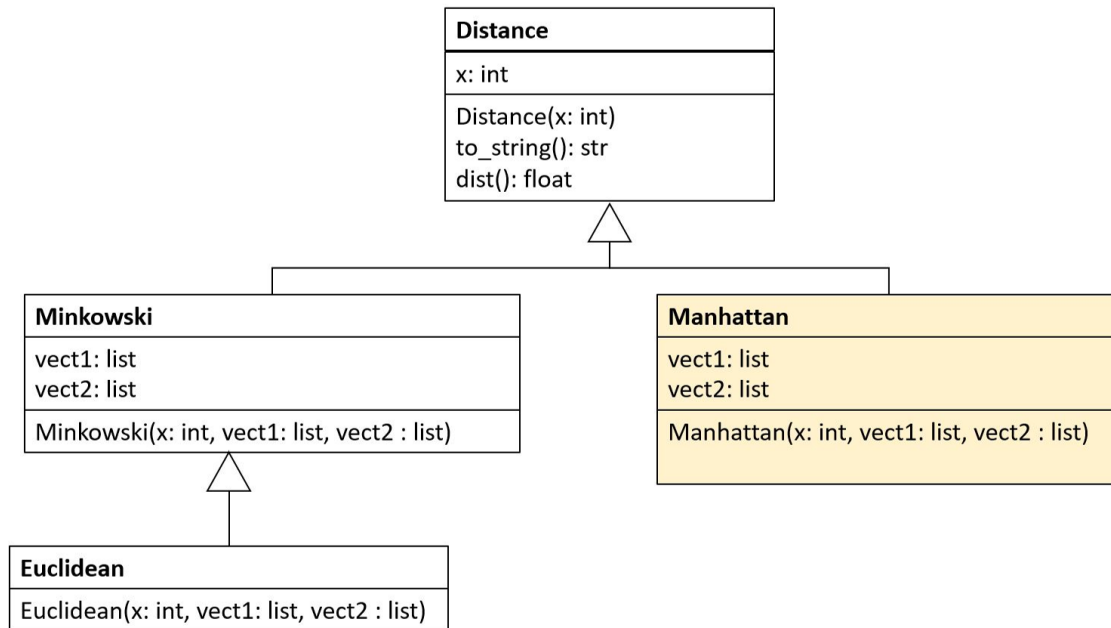
Returns the minkowski distance of the 2 vectors as a float. The output should be rounded to 4 decimal places. The Minkowski distance is defined as, $(\sum_{i=1}^n (|x(i) - y(i)|)^p)^{1/p}$, where $p=2$. Also, x and y are the first and second vectors, respectively.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from a7_ex3 import Distance`.
- Use `super().some_method()` to access the `some_method` implementation of the superclass.

Exercise 5 – Submission: a7_ex5.py**15 Points**

You are given the same class hierarchy as in the previous exercise that models 1D vectors.



In this exercise, you have to implement the concrete subclass **Manhattan**, which represents a city-block distance and extends the base class **Distance**. The class has the following additional instance attributes:

- **vect1: list**
The first vector.
- **vect2: list**
The second vector.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- **__init__(self, x: int, vect1: list, vect2: list)**
Sets both instance attributes (in addition to the attributes of the base class **Distance**).
- **to_string(self) -> str**
Returns a string representation of the form "Manhattan: the number of vectors =<x_value>, vector_1=<vect1_value>, vector_2=<vect2_value>", where <?_value> represents the value of the corresponding attribute.
- **dist(self) -> float**

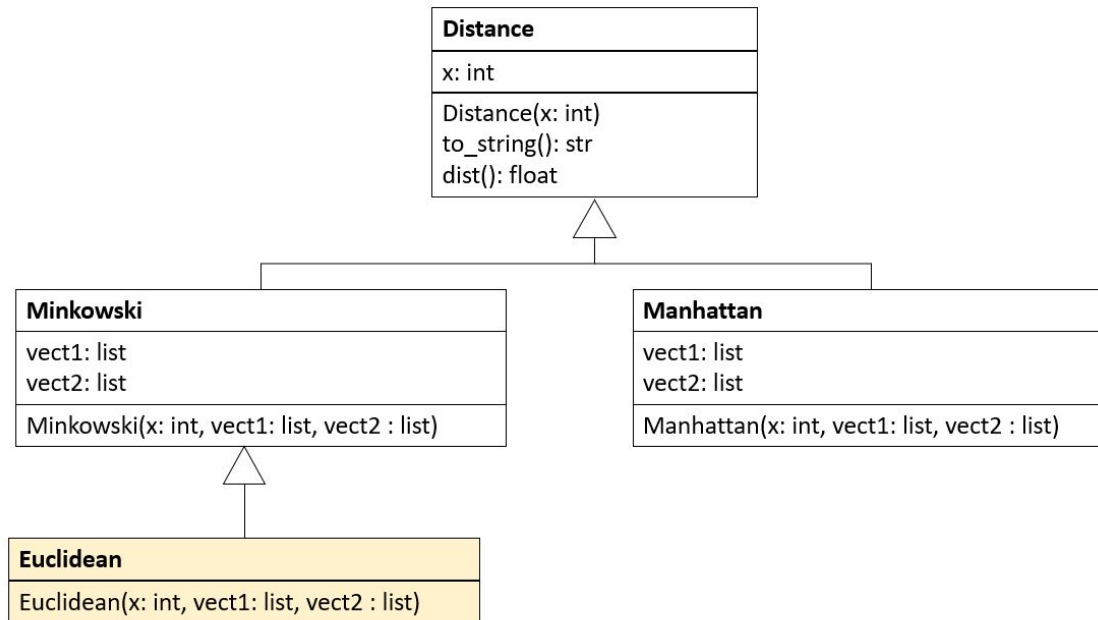
Returns the manhattan distance of the 2 vectors as a float. The output should be rounded to **4 decimal places**. The manhattan distance is defined as, $\sum_{i=1}^n |x(i) - y(i)|$, where **x** and **y** are the first and second vectors, respectively.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from a7_ex3 import Distance`.
- Use `super().some_method()` to access the `some_method` implementation of the superclass.

Exercise 6 – Submission: a7_ex6.py**15 Points**

You are given the same class hierarchy as in the previous exercise that models 1D vectors.



In this exercise, you have to implement the concrete subclass **Euclidean**, which represents a Euclidean distance and extends the base class **Minkowski**. The class has *no* additional instance attributes.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- `__init__(self, x: int, vect1: list, vect2: list)`

Sets both instance attributes (in addition to the attributes of the base class **Distance**). The **vector** parameters only exist for user convenience. Internally, the attributes **vect1** and **vect2** are fed to the parent class **Minkowski**.

- `to_string(self) -> str`

Returns a string representation of the form "Euclidean: the number of vectors =<x_value>, vector_1=<vect1_value>, vector_2=<vect2_value>", where <?_value> represents the value of the corresponding attribute.

- `dist(self) -> float`

Returns the Euclidean distance of the 2 vectors as a float. The output should be rounded to 4 decimal places. The euclidean distance is defined as, $\sqrt{\sum_{i=1}^n (x(i) - y(i))^2}$, where **x** and **y** are the first and second vectors, respectively.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from a7_ex4 import Minkowski`.
- The above requirement “reuse code from superclasses to avoid unnecessary code duplication” is especially relevant in this exercise. Depending on your implementation, it might be that you do not need to override any methods of the superclass `Minkowski`.

Combined Examples for Exercises 3, 4, 5 and 6

Example program execution:

```
vect1 = [1,2,3]
vect2 = [4,5,6]
d = Distance(2)
print(d.to_string())
try :
    d.dist()
except NotImplementedError:
    print("NotImplementedError")

k = Minkowski(2, vect1, vect2)
print(k.to_string())
print("Minkowski distance: ", k.dist())

m = Manhattan(2, vect1, vect2)
print(m.to_string())
print("Manhattan distance: ", m.dist())

e = Euclidean(2, vect1, vect2)
print(e.to_string())
print("Euclidean distance: ", e.dist())
```

Example output:

```
Distance: the number of vectors =2
NotImplementedError
Minkowski: the number of vectors =2, vector_1=[1, 2, 3], vector_2=[4, 5, 6]
Minkowski distance: 5.1962
Manhattan: the number of vectors =2, vector_1=[1, 2, 3], vector_2=[4, 5, 6]
Manhattan distance: 9.0000
Euclidean: the number of vectors =2, vector_1=[1, 2, 3], vector_2=[4, 5, 6]
Euclidean distance: 5.1962
```