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

The only module you are allowed to use is `math` in exercise 1.

**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.

## Exercise 1 – Submission: `a8_ex1.py` 40 Points

Create a class `Angle` that converts angles in degrees to radians and vice-versa. The class has the following instance attributes:

- `degree: float`

  Represents the angle in degrees.

- `radian: float`

  Represents the angle in radians.

The class has the following instance methods:

- `__init__(self, degree: float = None, radian : float = None)`

  Sets the instance attributes.

  - If only `degree` specified, the `radian` attribute should be assigned with the `deg_to_rad`-method (see below).
  - If only `radian` specified, the `degree` attribute should be assigned with the `rad_to_deg`-method (see below).
  - If both arguments are specified, check with `consistency`-method (see below) if both values correspond to the same angle.
  - If neither argument is specified, raise `ValueError("Either degree or radian must be specified.")`.

- `consistency(self)`

  Checks if the `degree` and `radian` attributes correspond to the same angle. Use the `math.isclose()` to verify the consistency. If `False`, raise `ValueError("Degree and radian are not consistent.")`.

- `__eq__(self, other)`

  If `other` is an `Angle` instance, `True` is returned if both attributes `degree` and `radian` of `other` are equal to the ones of `self`, `False` otherwise. If `other` is not an instance of `Angle`, `NotImplemented` is returned. Use the `math.isclose()` to check the equalities.

- `__repr__(self)`

  Returns the following string format: `"Angle(degree=<degree>, radian=<radian>)"`, where `<degree>` is the angle in degrees and `<radian>` is the angle in radians. Both values should be shown with 3 decimals.

- `__str__(self)`

  Returns the following string format: `"<degree> deg = <radian> rad"`, where `<degree>` is the angle in degrees and `<radian>` is the angle in radians. Both values should be shown with 2 decimals. Example: `"90.00 deg = 1.57 rad"`

- `__add__(self, other)`

If `other` is an instance of `Angle`, adds `other` to `self` and returns a new `Angle` object with the result of this addition. Does the addition both on the `<degree>` and `<radian>`. Otherwise, returns `NotImplemented`.

- `__iadd__(self, other)`

  If `other` is an instance of `Angle`, adds `other` to `self` in-place and returns `self`. Does the addition both on the `<degree>` and `<radian>` and check consistency with `consistency`-method. Otherwise, returns `NotImplemented`.

Moreover, add the following static methods (`@staticmethod`):

- `deg_to_rad(degree)`

  Converts an angle from degrees to radians with $degree * (\pi/180)$. Use `math.pi` for $\pi$.

- `rad_to_deg(radian)`

  Converts an angle from radians to degrees with $radian * (180/\pi)$. Use `math.pi` for $\pi$.

- `add_all(angle: Angle, *angles: Angle)`

  Adds `angle` and all angles in `*angles` together and returns a new `Angle` object containing this sum. None of the input arguments must be changed, i.e., all angles specified by `angle` and `*angles` must remain the same.

Example execution of the programme: Output[a]:

```
a1 = Angle(degree=45)
a2 = Angle(radian=math.pi/4)
a3 = Angle(30, math.pi/6)

print(a1)                                   45.00 deg = 0.79 rad
print(a2.__repr__())                        Angle(degree=45.000, radian=0.785)
print(repr(a3))                             Angle(degree=30.000, radian=0.524)

print(a1 == a2)                             True
print(a1 + a2)                              90.00 deg = 1.57 rad
a1 += a3
print(a1)                                   75.00 deg = 1.31 rad

sum_angle = Angle.add_all(a1, a2, a3)
print(sum_angle)                            150.00 deg = 2.62 rad

try:
    a4 = Angle()
except ValueError as e:
    print(e)                                Either degree or radian must be specified.
try:
    a5 = Angle(degree=45, radian=1)
except ValueError as e:
    print(e)                                Degree and radian are not consistent.
```

[a] Empty lines are shown here just for clarity.

## Exercise 2 – Submission: `a8_ex2.py` 30 Points

Create a class `Power` that represents an exponent. The class has the following instance attribute:

- `exponent: float`

  Represents the exponent value.

The class has the following instance methods:

- `__init__(self, exponent)`

  Sets the instance attribute `exponent`. If `exponent` is not numerical raise `TypeError("The exponent must be a numerical value.")`.

- `__call__(self, x)`

  Returns `x` to the power of `exponent`. If `x` is not numerical raise `TypeError("Input must be a numerical value.")`

- `__mul__(self, other)`

  If `other` is a numerical value, adds `other` to `exponent` of `self` and returns a new `Power` object with the result of this addition. If `other` is another instance of `Power`, adds the exponents from `self` and `other` and returns a new `Power` object. Otherwise, returns `NotImplemented`.

Additionally, create a daughter class `Square` for which `exponent=2`.

Example execution of the programme:

```
x = 3
square = Square()
cube = Power(3)
print(square.exponent, square(x))
print(cube.exponent, cube(x))

m1 = square * 2
print(m1.exponent, m1.__call__(x))
m2 = square * cube
print(m2.exponent, m2.__call__(x))

try :
    square("foo")
except TypeError as e:
    print(e)
try :
    Power("foo")
except TypeError as e:
    print(e)
```

Output[a]:

```
2 9
3 27

4 81
5 243

Input must be a numerical value.

The exponent must be a numerical value.
```

[a] Empty lines are shown here just for clarity.

### Exercise 3 – Submission: `a8_ex3.py` 30 Points

Create a class `StandardScaler` that standardizes features by removing the mean ($\mu$) and scaling to unit standard deviation ($\sigma$). The transformation of each feature is given by $z = (x - \mu)/\sigma$. The class has the following instance attributes:

- `mu: float`

  Represents the mean.

- `sig: float`

  Represents the variance.

The class has the following instance methods:

- `__init__(self)`

  Sets the instance attributes. Both attributes should be set by default to `None`.

- `fit(self, features : list)`

  Calculates the mean and standard deviation of the input features. The standard deviation should be calculates as $\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$. You can assume values in `features` to be numerical.

- `transform(self, features : list)`

  Returns a list of scaled input features based on the `mu` and `sig` attributes. If `mu` or `sig` is `None`, raise `ValueError("Scaler has not been fitted.")`. You can assume values in `features` to be numerical.

- `fit_transform(self, features : list)`

  Combines the fit and transform steps and returns the fitted input features. You can assume values in `features` to be numerical.

- `__getitem__(self, key)`

  Enables index-based access to the attributes. If `key` is `0` value of `mu` is returned and `key` is `1` value of `sig` is returned. If `key` is out of range raise `IndexError("Index out of range")` and if `key` not of type `int` `TypeError("Indices must be integers")`.

Example execution of the programme:

Output[a]:

```
feats1 = [0,2,4,6,8,10]
feats2 = [1,3,5,7,9]

s = StandardScaler()
print(s.mu, s.sig)                          None None
s.fit(feats1)
print(s[0], s[1])                           5.0 3.7416573867739413
feats1_scaled = s.transform(feats1)
print(feats1_scaled)                        [-1.3363062095621219, -0.8017837257372732, ...]
feats2_scaled = s.transform(feats2)
print(feats2_scaled)                        [-1.0690449676496976, -0.5345224838248488, ...]

s = StandardScaler()
feats2_scaled = s.fit_transform(feats2)
print(feats2_scaled)                        [-1.2649110640673518, -0.6324555320336759, ...]
print(s[0], s[1])                           5.0 3.1622776601683795

s = StandardScaler()
try:
    s.transform(feats2)
except ValueError as e:
    print(f"{type(e).__name__}: {e}")       ValueError: Scaler has not been fitted.
try:
    print(s["foo"])
except TypeError as e:
    print(f"{type(e).__name__}: {e}")       TypeError: Indices must be integers
try:
    print(s[2])
except IndexError as e:
    print(f"{type(e).__name__}: {e}")       IndexError: Index out of range
```

[a] Empty lines are shown here just for clarity.