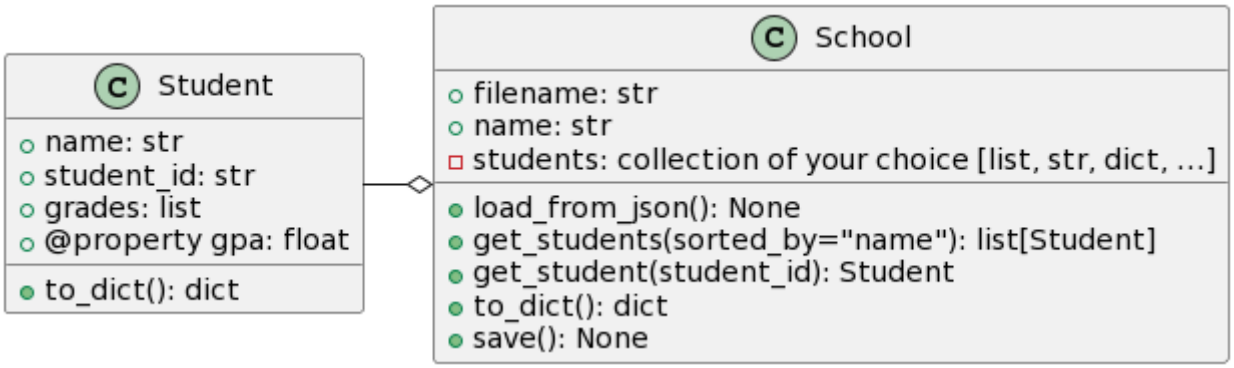
Lab: student management app with Flask

In this assignment, we are going to build a web app that allows visitors to view a list of students in a given school, and to view and manage their grades and GPA. This is a guided lab (not graded).

Create the models for your application

Your application will work with two classes: `School` and `Student`.



READ THE TESTS!

Create the folder structure

Your classes will live in the `models` Python package. Create that folder, and create an emtpy `__init__.py` file in it.

`Student`

The `Student` constructor takes 3 arguments:

- the name of the student (no default value)
- the student ID (no default value)
- a list of grades

These arguments are stored as attributes on your instance. If no list of grades is provided, the grades list should be an empty list.

> ⚠️ Be careful when using default arguments. **DO NOT USE `[]` as a default value!** Instead, you can do:
>
> ```
> def my_func(my_list=None):
>     if my_list is None:
>         my_list = []
>         # Do something with my_list
> ```

The `to_dict` method allows you to get a dictionary version of your instance. It will be very useful later.

`School`

The `School` constructor takes one argument: the name of a JSON file to load school information from. This filename is stored as the `filename` attribute.

Data is loaded with the `load_from_json` method. This method must open and load the JSON file, and create as many `Student` instances as necessary. The student instances will be saved in a **private variable** of your choice (could be `_students`, but does not have to, because it is private). The **type** of this variable is also up to you. You may choose a list, but it could be wiser to choose something else. **The implementation is up to you!**

The class also has the following methods:

- `get_students(sorted_by="name")`: returns a **LIST** of students. The `sorted_by` parameter defines how the list is sorted:
  - `name`: student list sorted by name ascending
  - `gpa`: student list sorted by gpa **DESCENDING** (i.e. highest GPA first)
- `get_student(student_id)`: returns a single `Student` whose student ID matches the parameter. If no student matches, return `None`.
- `to_dict`: returns a dictionary version of your school instance. Use the `Student.to_dict` method!
- `save`: transforms the instance into JSON and saves it to the filename attribute.

Advanced sorting

You can sort a collection using `collection.sort()` (will sort in place) or `sorted(collection)` (will **RETURN** a sorted copy of your collection). These functions take additional parameters:

- `key`: must be a function or callable that **RETURNS** the value used for sorting
- `reverse`: a boolean, used to define the sorting direction (`True` for ascending, `False` for descending)

You may use lambda functions for the sorting `key`. A good alternative is to use the `operator` module.

With the following class:

```
class Example:
    def __init__(self, name, number):
        self.name = name
        self.number = number
```

And a list `my_list` made of instances of the `Example` class, you can use:

```
import operator
sorted(my_list, key=operator.attrgetter("name"))                    # will sort
by name (ascending, default)
```

```
sorted(my_list, key=operator.attrgetter("number"), reverse=True)        # will sort
by number descending
```

`attrgetter` only works with **attributes** (or properties). You can also "replace" it with a lambda function (= a function defined inline). This can be useful if you want to use the output of a method to do the sorting.

```
sorted(my_list, key=lambda obj: obj.get_name())
sorted(my_list, key=lambda obj: obj.get_number(), reverse=True)
```

Create the Flask application

Create a new `app.py` file, and create an empty Flask app:

```
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
    app.run(debug=True)
```

You should be able to run this application by running: `python app.py` in the terminal. Your Flask application will be available on your local computer, typically on the following URL: http://127.0.0.1:5000/.

Create a route for the homepage

Add a route for the URL `/`:

```
from flask import Flask, render_template

@app.route("/")
def home():
    return render_template("home.html")
```

In order for this to work, Flask needs to have an HTML template file called `home.html` in the `templates` folder. Create that folder and an HTML file. Make sure you can access your HTML file in Flask.

Display information from your models in your homepage

Make the homepage display a list of students for the school with filename `bcit.json`.

```python
@app.route("/")
def home():
    school = School("bcit.json")
    return render_template("list.html", school=school)
```

This will make the `school` variable available in the HTML template. Add HTML bits to loop through all students (ordered by name) and display them as a list in the HTML page, with their GPA.

> Use `for` loops in the template:
>
> ```
> {% for student in school.get_students() %}
>   {{ student.name }}
> {% endfor %}
> ```

Create a route to display information about a student with a parameter

Flask routes can take parameters: values that change depending on the HTTP request.

To add a new route which takes a `student_id` parameter, you can do:

```python
@app.route("/student/<string:student_id>")
def student(student_id):
    # do something with student_id
```

If you try to access http://127.0.0.1:5000/student/A01201234, Flask will run the function `student`, and `student_id` will have the value `A01201234`.

Add the Python code to display a template that displays a template for a single student given a `student_id`. You will need to create a new HTML template, and send it the relevant `Student` object. Make sure your view displays all the grades of the student.

Connect the pages together

Change your homepage template to add links to the list of students, so that clicking on a student displays all their grades.

> ⚠️ Do not write links manually! Use `url_for`.

URL reverse lookups

Flask can provide you with the URL for a specific view with arguments. You should use this in templates to make sure your application is independent from your URL structure.

```
<a href="{{ url_for('home') }}">Homepage</a>
```

You can also use parameters:

```
<a href="{{ url_for('student', student.name) }}">Homepage</a>
```

Going further: template scaffolding

Instead of repeating the HTML structure in each HTML template, it is better to build a *base* template that defines blocks. Subtemplates can extend from the base template and define their own blocks. For instance:

- in `base.html`:

```
<html>
    <body>
        {% block content %}
        {% endblock %}
    </body>
</html>
```

- in `detail.html`:

```
{% extends "base.html" %}

{% block content %}
    <h1>{{ instance.name }}</h1>
{% endblock %}
```

The `H1` tag will be added to the block in the base template.

A `base.html` template for Bootstrap 5 is available on D2L. Make sure you remove the `.txt` extension of the file.