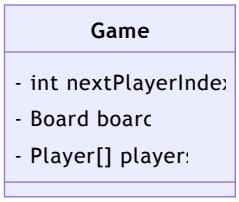
TicTacToe - Data Models

Data models

A data model is a conceptual representation of the data structures that are required by an application. A data model is the code equivalent of a UML class diagram of the entities. It is used to define the logical structure and essentially determines in which manner data can be stored, organized, and manipulated. There can be different sets of data models depending on the abstraction level at which they are used. We shall talk a lot more about this when we discuss the three-layered architecture.
For tic-tac-toe, we shall be creating classes which contain the state (i.e. the attributes) and the behavior (i.e. the methods) of the entities. There are multiple ways to define data models in Python.

Standard classes

Classes are defined in Python using the class keyword and the attributes are initialised in the `__init__` method. As mentioned before, the `__init__` method is an example of a dunder function. The `__init__` method is called when an object of the class is created. The `self` keyword is used to refer to the object itself. The `self` keyword is similar to the `this` keyword in Java and C++.
Let us start by creating a simple class for the `Game` entity




```
class Game:
    def __init__(self, board, players):
        self.current_player_index = 0
        self.board = board
        self.players = players
```

Python is a dynamically typed language. This means that the type of a variable is inferred at runtime. This is in contrast to statically typed languages like Java and C++ where the type of a variable is known at compile time, and we do not need to specify the type of the attributes of a class. However, it is a good practice to specify the type of the attributes using type hints. This helps in understanding the code better and also helps in catching bugs early. The type hints are not enforced by the Python interpreter, but there are tools like **mypy** (https://mypy.readthedocs.io/en/stable/) which can be used to enforce type hints.

```
from typing import List
class Game:
    def __init__(self, board: Board, players: List[Player]):
        self.current_player_index = 0
        self.board = board
        self.players = players
```

Before creating the `Board` class, let us see a different and more concise way of defining classes in Python.

Dataclasses

The `dataclasses` module was introduced in Python 3.7. It provides a decorator and functions for automatically adding generated special methods such as `__init__` and `__repr__` to user-defined classes. It also provides a function decorator which can be used to add generated special methods to existing classes.

> Decorators are a way to dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. This is ideal when you need to extend the functionality of functions that you don't want to modify. Decorators are prefixed with an `@` symbol and are placed above the function definition. They are similar to annotations in Java.

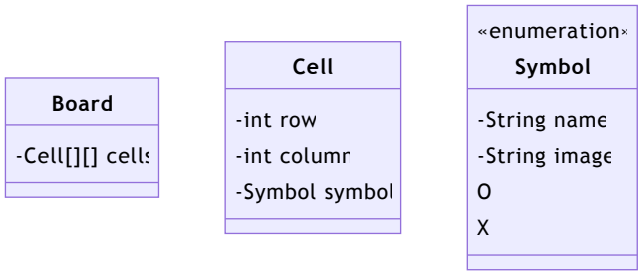
Let us see how we can use the `dataclasses` module to create the `Game` class.

```
from dataclasses import dataclass

@dataclass
class Game:
    current_player_index: int
    board: Board
    players: List[Player] = field(default_factory=list)
```

Defining the class using the `dataclass` decorator is much more concise. You add the attributes of the class along with their types.
The `dataclass` decorator automatically adds the `__init__` method to the class. The `__init__` method is used to initialise the attributes of the class.
The `dataclass` decorator also adds the `__repr__` method to the class. The `__repr__` method is used to return a string representation of the object. The `__repr__` method is called when the `repr()` function is called on the object. The `__repr__` method is also called when the object is printed. The `__repr__` method is similar to the `toString()` method in Java.
The `field` method is used to provide metadata about the attributes of the class. Here we are using the `default_factory` parameter to specify that the default value of the `players` attribute should be an empty list. The `default_factory` parameter is used to specify a callable that will be called without arguments to initialize the attribute. The `default_factory` parameter is similar to the `default` parameter in Java.

The Board class

Let us now look at creating the `Board` and the associated classes.



We start with the `Symbol` enum class. To create an enum in Python, we use the `enum` module. The `enum` module was introduced in Python 3.4. Enums are a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.

```
from enum import Enum

class Symbol(Enum):
    O = 1
    X = 2
```

The right hand side of the assignment is the value of the enum member that should be a unique integer. The left hand side is the name of the enum member. The name of the enum member is also the string representation of the enum member. The string representation of the enum member can be accessed using the `name` attribute of the enum member. Let us create the `Cell` class next. The `Cell` class has three attributes - `row`, `column` and `symbol`.

```
from dataclasses import dataclass

@dataclass
class Cell:
    row: int
    column: int
    symbol: Symbol
```

Now to create the `Board` class, we could create a simple class with a 2D array of `Cell` objects and a similar constructor

```
from dataclasses import dataclass
from typing import List

@dataclass
class Board:
    cells: List[List[Cell]]
```

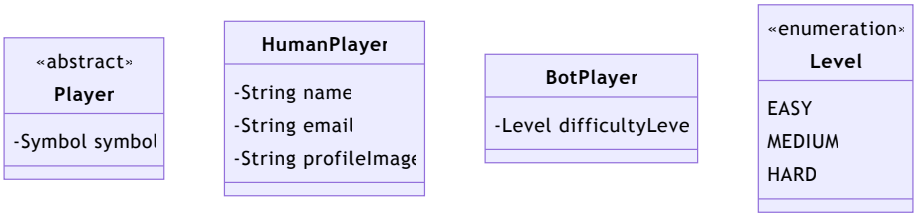
> The `typing` module was introduced in Python 3.5. It provides runtime support for type hints. The `typing` module defines a number of aliases for common types. For example, `List` is an alias for `list`, `Dict` is an alias for `dict`, `Tuple` is an alias for `tuple`, etc. The `typing` module also defines a number of generic types. For example, `List[int]` is a list of integers, `Dict[str, int]` is a dictionary with string keys and integer values.

However, we can do better. Instead of having a constructor which accepts a 2D array of `Cell` objects, we can have a constructor which accepts the size and then constructs the 2D array of `Cell` objects. To do this, we can use the `__post_init__` method. The `__post_init__` method is called after the `__init__` method. The `__post_init__` method is used to perform any additional initialisation. Also, we will use the `field` function from the `dataclasses` module to provide metadata about the attributes of the class. Here we will use the `init` parameter to specify that the attribute should not be included in the `__init__` method. This is because we will be initialising the attribute in the `__post_init__` method.

```
from dataclasses import dataclass, field
from typing import List

@dataclass
class Board:
    cells: List[List[Cell]] = field(init=False)

    def __post_init__(self):
        self.cells = self.initialize_cells()

    def initialize_cells(self) -> List[List[Cell]]:
        cells = []
        for row in range(self.size):
            row_cells = [Cell(row, column) for column in range(self.size)]
            cells.append(row_cells)
        return cells
```

The Player classes

Now let us create the player classes as below:



Let's start with the `Player` class. The `Player` class is an abstract class. To create an abstract class in Python, we use the `ABC` class from the `abc` module. The `ABC` metaclass is used to create abstract base classes. An abstract base class is a class that cannot be instantiated and has abstract methods that must be implemented by its subclasses. The `ABC` metaclass is similar to the `abstract` keyword in Java.

```
from abc import ABC

class Player(ABC):
    symbol: Symbol
```

The `HumanPlayer` class has three attributes - `name`, `email` and `profile_image`.

```
from dataclasses import dataclass

@dataclass
class HumanPlayer(Player):
    name: str
    email: str
    profile_image: str
```

The profile image can cause memory issues since one player can have multiple games and for each game a new object of the `HumanPlayer` class will be created. To avoid this, we can use the flyweight pattern.
The flyweight pattern is a structural design pattern that allows sharing objects to support large numbers of fine-grained objects efficiently. It is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects. To implement the flyweight pattern, we divide the class into two parts - the intrinsic state and the extrinsic state. The intrinsic state is the state that is shared across objects. The extrinsic state is the state that is unique to each object. The intrinsic state is stored in a flyweight object and the extrinsic state is stored in a context object. The context object is passed to the flyweight object when the flyweight object is created.
So we have the `User` class as the flyweight object and the `HumanPlayer` class as the context object. The `User` class will contain the intrinsic state and the `HumanPlayer` class will contain the extrinsic state.

```
from dataclasses import dataclass

@dataclass
class User:
    name: str
    email: str
    profile_image: str

@dataclass
class HumanPlayer(Player):
    user: User
```

The `BotPlayer` class has one attribute - `difficulty_level` . Let us create the `Level` enum class first.

```
from enum import Enum

class Level(Enum):
    EASY = 1
    MEDIUM = 2
    HARD = 3
```

Now let us create the `BotPlayer` class.

```
from dataclasses import dataclass

@dataclass
class BotPlayer(Player):
    difficulty_level: Level
```

Conclusion

We looked at the different ways of defining data models in Python i.e. using standard classes and dataclasses. We used the `dataclasses` module to create the data models for the tic-tac-toe game. The next time we will look at adding behavior to the data models using methods.