What We’ll Be Seeing Today

- Short recap
- Student enrollment system
- Popcorn Time
- Ex 9
Programming Paradigms

- **Procedural**: Modules and data structures with procedures/methods that operate on them.
- **Object Oriented**: Encapsulate our data/state/behavior inside objects, these objects communicate among themselves.
- **Functional**: Functions, closures, recursions etc. Treats everything as mathematical formulas.
Too Many Options?

- Python allows us to utilize every one of the programming paradigms mentioned! This allows the programmer to:
  - Choose the paradigm that best suits his program.
  - Mix paradigms together.
  - Switch between paradigms if needed.
Until now we have mostly seen algorithms and basic python syntax (well and some basic OOP).

OOP on the other hand is a concept!

- Meaning it is a way for us to create programs.
- We can compare OOP to the architect designing a building: OOP will provide the plan, it’s building blocks will be Python and the algorithms we’ve seen.
In python every thing we’ve seen is actually an object!

- A list is an object.
- A string is an object.
- ...

For example calling `list()` will create an empty list.
System Design

- Last tirgul you saw a bottom up approach
  - We first designed all the elements in our game only later to design the game itself

- Today we are going to see the opposite
  - Design the system and than implement it

- There is no “right” way, it is mostly application dependent.
We want to design a system where students could register to courses.

What is the most basic expected behavior from such a system?
- List all courses
- Enroll into a course
- Withdraw from a course
We need to think about the properties of our enrollment class:
- Available courses?
- Map students to courses?
- Keep track of registered students?

Do we need to think on how we design students/courses at this stage?
class EnrollmentSystem:
    def __init__(self):
        self._courses = {}
        self._students = {}

    def add_student(self, student):
        # How can we map a student to a student object?
        self._students[student.get_id()] = student

    def add_course(self, course):
        # How can we map a course to a course object?
        self._courses[course.get_id()] = course

What are our assumptions?
What could go wrong here?
class EnrollmentSystem:
    def __init__(self):
        self._courses = {}
        self._students = {}

    def get_available_courses(self):
        courses = []
        for course_id, course in self._courses.items():
            if not course.is_full():
                courses.append( ( course_id, course.get_name() ) )
        return courses

What are our assumptions?
What about registration?

class EnrollmentSystem:
    def __init__(self):
        self._courses = {}
        self._students = {}

    def register_student(self, student, course_id):
        if course_id in self._courses:
            registered = self._courses[course_id].register(student.get_id())
            if registered:
                student.add_course(course_id)
        return registered

What are our assumptions?
What did we assume so far?

- Course
  - get_id()
  - is_full()
  - register( sid )

- Student
  - get_id()
  - add_course( cid )

- Do the EnrollmentSystem care how we implement these methods?
Student Class

- Lets implement our basic student class

```python
class Student:
    def __init__(self, student_id):
        self._enrolled_courses = set()
        self._id = student_id

    def get_id(self):
        return self._id

    def add_course(self, course_id):
        self._enrolled_courses.add(course_id)
```

Let's implement our basic student class
class Course:
    def __init__(self, course_id, course_capacity):
        self._enrolled_students = set()
        self._id = course_id
        self._capacity = course_capacity

    def get_id(self):
        return self._id

    def register(self, student_id):
        if student_id not in self._enrolled_students:
            self._enrolled_students.add(student_id)
            return True
        return False

    def is_full(self):
        return self._capacity == len(self._enrolled_students)
Putting things together

- We want to create an actual program using our enrollment system

- A user should be able to:
  - Add courses to the system
  - Register users to courses
Our main loop

from student import Student
from course import Course
from system import EnrollmentSystem

def get_user_selection():
    return int(input("Enter selection: "))

def main_loop():
    es = EnrollmentSystem()
    while True:
        option = get_user_selection()
        perform_selection(option, es)

...  
def perform_selection(option, es):
    if option == 1:
        print_courses(es)
    elif option == 2:
        register_new_student(es)
    elif option == 3:
        create_new_course(es)
    elif option == 4:
        register_student_to_course(es)
    else:
        print("Please insert a valid option")
What’s next?

- Well a lot is missing from the system
  - Actual implementation of the methods 😊
  - Withdrawing from a course
  - Removing a course
  - Course info?
  - …

- We can always come back and extend our system
To OOP or not to OOP? – A word on being cautious

- Not everything should be solved by new objects.
  - You should always think about the overhead of working with them.

- For example, should you create a class to represent 2D points or could you just use tuples?
Movie Player

We will now consider the parts needed to build a movie player application such as PopcornTime.

- It is an application that allows you to stream media from torrents to your computer.

We will try to analyze the parts needed for this system to work, and we will implement some of them – the rest you could implement by the end of the course!
Why A Movie Player?

- Although building a movie player in procedural programming is possible – it would be too hard and not generic enough.

- Each part of the player is independent of the other parts – what we care about is only the API they supply us!
An “Application Programming Interface” (API) is a way for us to expose the behavior of our objects without the actual way it was implemented.

Consider the list construct: The list object exposes a method called “sort” – does it matter which sort algorithm is used?

Most of the time what interest us is the final result.
PopcornTime Analysis
What do we need to represent for us to implement this (no user-interface for now)?

<table>
<thead>
<tr>
<th>Movie</th>
<th>User Preferences</th>
<th>Search Engine</th>
<th>Player</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Title</td>
<td>• Usage history</td>
<td>• By actors</td>
<td>• <strong>Play a movie</strong></td>
</tr>
<tr>
<td>• Length</td>
<td>• Favorites</td>
<td>• By category</td>
<td>• Continue a paused movie</td>
</tr>
<tr>
<td>• Language(s?)</td>
<td>• Saved for later</td>
<td>• By year</td>
<td>• Display subtitles</td>
</tr>
<tr>
<td>• Thumbnail</td>
<td>• Login?</td>
<td>• By director</td>
<td>• Change subtitles</td>
</tr>
<tr>
<td>• Subtitles</td>
<td>• Communication type</td>
<td>• …</td>
<td>• Change audio language</td>
</tr>
<tr>
<td>• Qualities</td>
<td>• …</td>
<td></td>
<td>• Cast to Chromecast?</td>
</tr>
<tr>
<td>• Was seen?</td>
<td></td>
<td></td>
<td>• …</td>
</tr>
<tr>
<td>• Rating (from where?)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Actors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Categories</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• …</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

24
Like our system example this program will be no different – only more complex.

We will need to create classes to represent a movie (and its properties), movie player, settings class and so on.

This is the idea of OOP, divide our program structure into objects with some sort of relationship between them.
First let’s think about what we are going to do with our movie:

- Watch it
- Display it’s info
- Search for it
- …

Do we need to keep all of the data? Or just it’s location?

Should we ask for this data each time? Or should we cache it?

Which pieces of information are we going to search on?
Designing An API

- One of the most common operations we will need from our movie is to query about its data – it does not change (for now).

- So we will need to have access to:
  - Actors, year, description etc…

- This should be done using “getters” – they get information from the object.
  - There are also “setters” which set/update new information.
Our Movie Class

class Movie:
    def __init__(self, ???):
        ???
    def get_actors(self):
        pass
    def get_description(self):
        pass
    def get_year(self):
        pass
    def get_length(self):
        pass
    def get_rating(self):
        pass
    def get_title(self):
        pass
    def get_torrent_link(self):
        pass
    def get_available_qualities(self):
        pass
    def get_available_subtitles(self):
        pass
Our Movie Class

- So each of the `get_<field>` methods body will probably just be: `return self.__<field>`

- This leaves us with the task of just deigning the `__init__` method
  - We will assume some magic API is given for us to use – by whom was it given? Later on in the course…
class MagicAPI:
    # Returns a list of strings, each string represents a movie in the
    # following format:
    # <title>_<year>_<description>_<thumb_link>_<torrent_url>
    def get_all_movies(): …
    
    # Returns the rating (float between 0-5) of a given movie
    def get_rating(movieName): …
    
    # Returns a list of strings, each string represents an actor:
    # <first name>_<last name>
    def get_actors(movieName): …

We will assume more magical possible operations as we go
To finish our movie class we will implement the `__init__` method and some other methods using our MagicAPI:

```python
class Movie:
    def __init__(self, movieStr):
        strAsList = movieStr.split('_')
        self.__title = strAsList[0]
        self.__year = strAsList[1]
        self.__description = strAsList[2]
        self.__thumbnailLnk = strAsList[3]
        self.__torrentLnk = strAsList[4]

    def get_actors(self):
        return MagicAPI.get_actors(self.__title)

    def get_rating(self):
        return MagicAPI.get_rating(self.__title)
```
Rethinking Our Design

- So we can now get a list of actors of a specific movie, but is that enough?
- What if we want to search movies by actors?
- Right now it would require us to query for the actors of every movie and only return those with specific actors.
- What if we want to find movies with several actors?
An Actor Class

class Actor:
    def __init__(self, name, movies = []):
        self.__first, self.__last = name.split('_')
        self.__movies = set()  # What is the limitation here?
        for movieName in movies:  # List of movie names
            self.__movies.add(movieName)

    def get_movies(self):
        return self.__movies

    def get_name(self):
        return self.__first + " " + self.__last
class Actor:

    ... 

    def appeared_in(self, movie):
        return movie in self.__movies

    def add_movie(self, movie):
        self.__movies.add(movie)
So we have an Actor and a Movie classes, but what can we do with them?

- We need some entity to manage everything for us!
- This entity should allow us to search over our movies and actors. Also, it should support adding new movies/actors to it’s existing knowledge base.
class MovieManager:
    def __init__(self):
        self.__movies = {}
        self.__actors = {}

        for movieStr in MagicAPI.get_all_movies():
            movie = Movie(movieStr)
            for actorName in movie.get_actors():
                if actorName not in self.__actors.keys():
                    self.__actors[actorName] = Actor(actorName)
                    self.__actors[actorName].add_movie(movie)
Movie Manager Class

Now we can think of many operations we could do with this data, we could:

- Query for movies starting with some string. Query for movies by actors.
- Find movies where one (or more) actors appear.
class MovieManager:

    ...  

    def find_movies_start_with(self, query):
        result = []
        for movieName in self.__movies.keys():
            if movieName.startswith(query):
                result.append( self.__movies.get(movieName) )
        return result

    def find_movies_with_actor(self, actor):
        return self.__actors(actor).get_movies()
class MovieManager:

    ...  

def find_movies_with_actors(self, actor1, actor2):
    result = []
    for movie in self.__actors.get(actor1).get_movies():
        if movie in self.__actors.get(actor2).get_movies():
            result.append(movie)
    return result


def find_movies_with_rating(self, rating):
    return [movie for movie in self.__movies.values() if movie.get_rating() == rating]
Is It Over?

- Well no, we are still missing much of the required functionality – but is it that difficult to achieve?
- If we continue to assume things about our MagicAPI class (ability to download a torrent) we could build our infrastructure.
- And then we are “only” left with the task of implementing our MagicAPI 😊
Ex 9
Ex 9 – Playing With Objects

- In this exercise you will need to implement the game of Asteroids!
- You will only be given a Screen class that handles all of the graphics for you
  - All of the actual design is left to you!
- We advise you to work on this exercise in pairs
Ex 9 – Game Loop

- Most of the time games (and programs) are event driven:
  - Move right/left if a key pressed.
  - Fire when requested to.
  - Reduce HP if got shot.
  - Etc…

- We will take a more sequential approach.
Ex 9 – Game Loop

- We can think that everything happens in the same time!
  - All objects (ship, asteroids, torpedoes) move one after the other.
  - Only then we fire/accelerate (if requested)
  - More in the exercise.

- **Note:** Try to implement it step by step (as described in the exercise description).
Ex9 – Working in pairs

- You should exchange ideas on how you think it is right to implement the tasks
  - You can even write things down

- Try and plan ahead

- Think about the division of the exercise
  - One codes and the other reviews?
  - Take turns?