Extended Introduction to Computer Science
CS1001.py

Chapter G
Lecture 14a

Intro to object oriented programming (OOP) and Data Structures

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Plan for the next lectures

- Intro to object oriented programming (OOP)

- Data Structures
  1. Linked Lists (today? Probably not)
  2. Binary Search Trees
  3. Hash tables
  4. Iterators and generators

Object Oriented Programming (OOP)

- OOP is a major theme in programming language design, starting with Simula, a language for discrete simulation, in the 1960s. Then Smalltalk in the late 1970s (out of the legendary Xerox Palo Alto Research Center, or PARC, where many other ideas used in today's computer environment were invented). Other "OOP languages" include Eiffel, C++, Java, C#, and Scala.

- Python supports object oriented style programming (maybe not up to the standards of OOP purists). We'll describe some facets, mostly via concrete examples. A more systematic study of OOP will be presented in Tochna 1, using Java.

Object Oriented Programming (OOP), cont.

- Entities in programs are modeled as objects. They represent encapsulations that have their own:
  1) attributes (also called fields), that represent their state
  2) methods, which are functions or operations that can be performed on them. Creation and manipulation of objects is done via their methods.

- The object oriented approach enables modular design. It facilitates software development by different teams, where each team works on its own object, and communication among objects is carried out by well defined methods' interfaces.

Classes and Objects

- We already saw that classes represent data types. In addition to the classes/types that are provided by python (e.g., str, list, int), programmers can write their own classes.

- A class is a template to generate objects. The class is a part of the program text. An object is generated as an instance of a class.

- As we indicated, a class includes data attributes (fields) to store the information about the object, and methods to operate on them.

Let’s think about classes we would like to implement …

What fields would they have?

What methods will they include?

Student Class - Executions

```
>>> s1 = Student("Donald", "Trump", 123456789)
>>> s1
<Donald, 123456789>


>>> s1.update_grade("CS1001", 91)
>>> s1.grades
{'CS1001': 91}
>>> s1.update_grade("HEDVA", 90)
>>> s1.update_grade("CS1001", 98) #he appealed
>>> s1.grades
{'HEDVA': 90, 'CS1001': 98}
>>> s1.avg()
94.0


>>> s2 = Student("Vladimir", "Putin", 888888888)
>>> s2.update_grade("Algebra", 95)
>>> s2.update_grade("CS1001", 100)
>>> print(s2, s2.grades, s2.avg())
<Vladimir, 888888888> {'CS1001': 100, 'Algebra': 95}  97.5
```

Building Class Student

```
class Student:

    def __init__(self, name, surname, ID):
        self.name = name
        self.surname = surname
        self.id = ID
        self.grades = dict()

    def __repr__(self): #must return a string
        return "<" + self.name + ", " + str(self.id) + ">"

    def update_grade(self, course, grade):
        self.grades[course] = grade


    def avg(self):
        s = sum([self.grades[course] for course in self.grades])
        return s / len(self.grades)
```

__init__and __repr__
are special standard methods,
with pre-allocated names.
More on this coming soon.

Student Class (cont.)

- The Student class has 4 fields: name, surname, id and a dictionary of grades in courses. These fields can be accessed directly, and values can be assigned to them directly.

- The methods (operations) of the class are:
  - __init__ used to create and initialize an object in this class
  - __repr__ used to describe how an object is represented (when printing such an object).
  - update_grade used to insert a new grade or update an existing one
  - avg returns the average of the student in all the courses

The constructor __init__

- __init__ is called when the class name is written, followed by parameters in ().

```
Student("Donald", "Trump", 123456789)
```

- The fields of the class we are defining exist because they are initialized in the `__init__` method.

- So the variable `s1.name` is the field named `name` in the object `s1`.

Who is self (or: who am I)?

- The first parameter of every method represents the current object (an object of the class which includes the method). By convention, we use the name self for this parameter.

- so `self.name` is the field named `name` in the current object.

- When calling a method, this parameter is not given explicitly as the first parameter, but rather as a calling object

Calling methods

- We have seen that we call a method by its full name, preceded by an object of the appropriate class, for example `s1.avg()`.

- But we can also call it using the name of the class (rather than a specific object). In this case the first parameter will be the calling object:

```
>>> s1 = Student("Donald", "Trump", 123456789)
>>> Student.update_grade(s1, "HEDVA", 90)
>>> Student.avg(s1)
90
```

Special Methods

- There are various special methods, whose names begin and end with __ (double-underscore). These methods are invoked (called) when specific operators or expressions are used.
- Following is a partial list. The full list and more details can be found at: https://diveintopython3.net/special-method-names.html

You Want…	So You Write…	And Python Calls…
to initialize an instance of class MyClass	x = MyClass()	x.__*init*__()
the "official" representation as a string	*print*(x)	x.__*repr*__()
addition	x + y	x.__*add*__(*y*)
subtraction	x - y	x.__*sub*__(*y*)
multiplication	x * y	x.__*mul*__(*y*)
equality	x == y	x.__*eq*__(*y*)
less than	**x < y**	x.__*lt*__(*y*)
for collections: to know whether it contains a specific value	k in x	x.__*contains*__(k)
for collections: to know the size	len(x)	**x.__len__()**
… (many more)		

Student Class, Defining Equality

```
>>> s1 = Student("Donald", "Trump", 123456789)
>>> s1
<Donald, 123456789>
>>> s2 = Student("Donald", "Trump", 123456789)
>>> s2
<Donald, 123456789>
>>> s1==s2
False #Hah??
```

Student Class, Defining Equality (2)

- Unless otherwise defined, Python compares objects by their memory address.
- __eq__ is a special method that determines when two objects (in this case) lines are equal.

```
def __eq__(self, other):
    assert isinstance(other, Student)
    return self.id == other.id
```

```
>>> s1 = Student("Donald", "Trump", 123456789)
>>> s2 = Student("Donald", "Trump", 123456789)
>>> s1 == s2 # __eq__ is called , same as s1.__eq__(s2)
True #☺
```

Information hiding (for reference only)

- One of the principles of OOP is information hiding: The designer of a class should be able to decide what information is known outside the class, and what is not. In most OOP languages this is achieved by declaring fields and methods as either public or private.

- In python, a field whose name starts with two _ symbols, will be private. It will be known inside the class, but not outside.

- A private field cannot be written (assigned) outside the class, and its value cannot be read (inspected), because its name is not known. The class then provides methods to access and modify the state of the object in the "legal" way.

OOP and Python

- Python provides the basic ingredients for OOP, including inheritance (that we will not discuss).
- However, we do not have the full safety that strict OOP languages have . “Private” fields are accessible with mangled names, a client may add a field to an object, etc. In short, there is no way to enforce data hiding in python, it is all based on convention.
- The language puts more emphasis on flexibility.
- In this course we will not use private fields to simplify the code (rather than adhere to OOP). This is the common style in python.
- The course Software 1 (in Java) places OOP at the center.

Designing classes in OOP

The recommended way to design a class is to

1) first decide what operations (methods) the class should support. This would be the API (Application Program Interface or contract) between the class designer and the clients (users).
2) then decide how to represent the state of objects (which fields), so that the operations can be performed efficiently, and implement a constructor (__init__).
3) then implement (write code for) the methods.

This way we can later change the representation (eg. change from Cartesian to Polar representation of points), while the client code is unchanged.