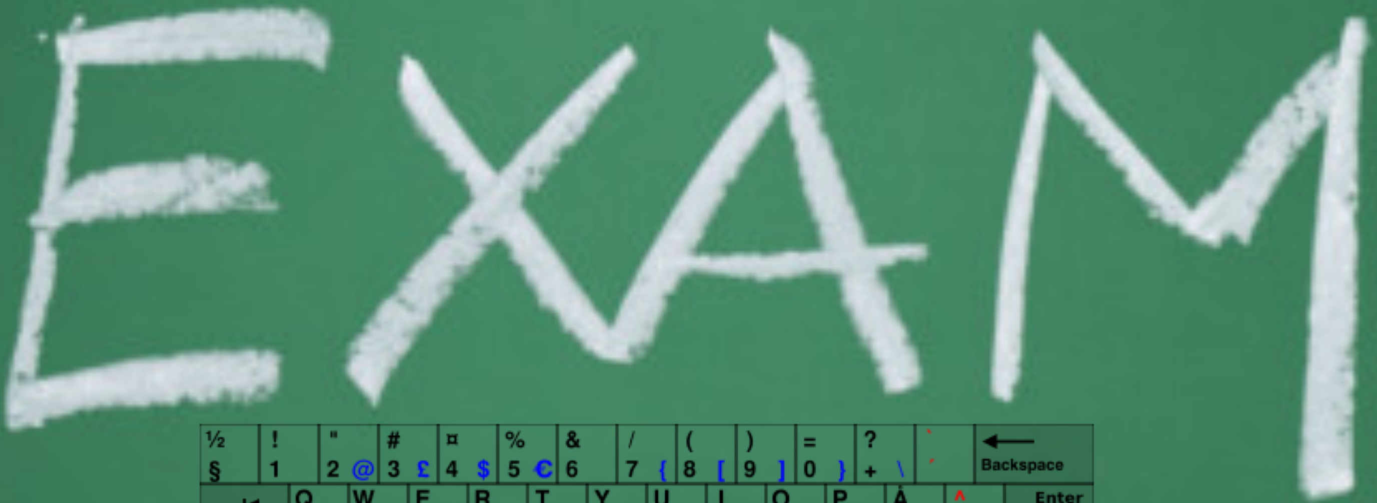
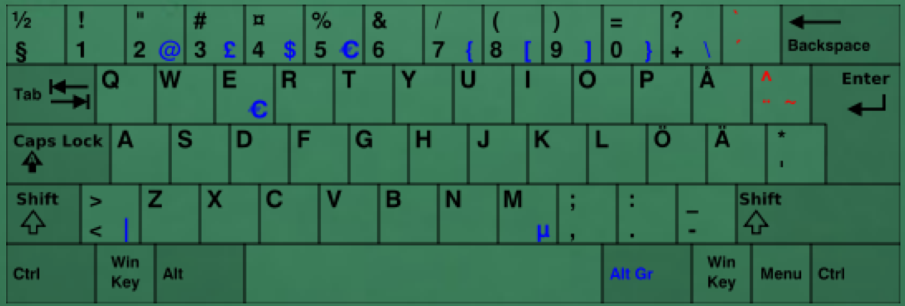
CS-A113 Basics in Programming Y1

9th Lecture
8.11.2021



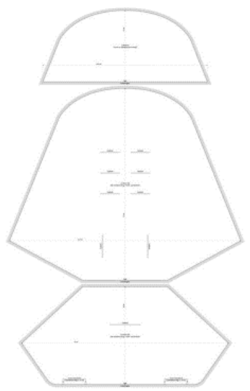


1. Don’t forget to register in SISU for the exam at least 1 week before the exam
2. Don’t forget to also register for your EXAM slot.
3. You find information on our myCourses page

- Onsite
- Finnish Keyboard
- No stuff (ID & water bottle without labels)
- First login with the login provided at the EXAM computer
- **Remember your Aalto login** – you need to login with this after

Topic Today: OO-Programming

Recap



Sewing Pattern

- You can build more than one object from it
- It describes the underlying structure
- It is not an object itself

A student has

- a name
- a student number
- courses they are enrolled in
- grades

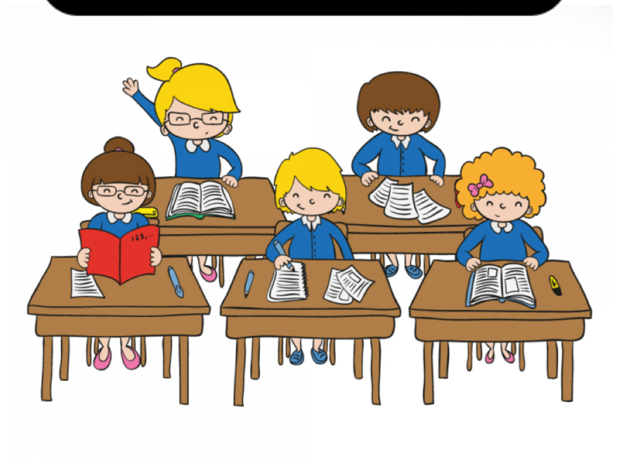
Recap:

```
class Student:

    def _ _ init_ _(self, myName, myNumber):
        self._ _ name = myName
        self._ _id = myNumber
        self._ _grades = [ ]
        self._ _courses = [ ]

    def add_course(self,course):
        self._ _courses.append(course)

main():
    student1 = Student(”Barbara”,123)
    student2 = Student(”Angelina”,564)
    studentRegistry = (student1,student2,student3)
    name = read_input()
    student1.add_course(”Basics in Programming”)
    student2.add_course(”Algorithms and Datastructures”)
```

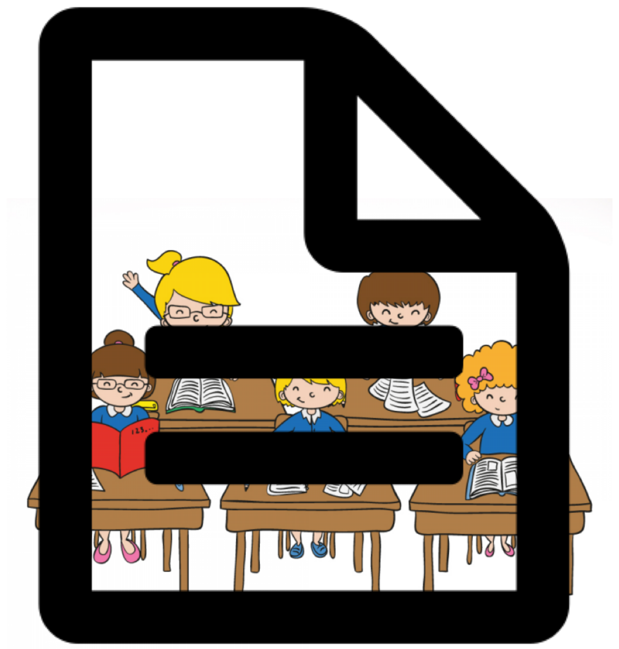
How to Make Large Programs?

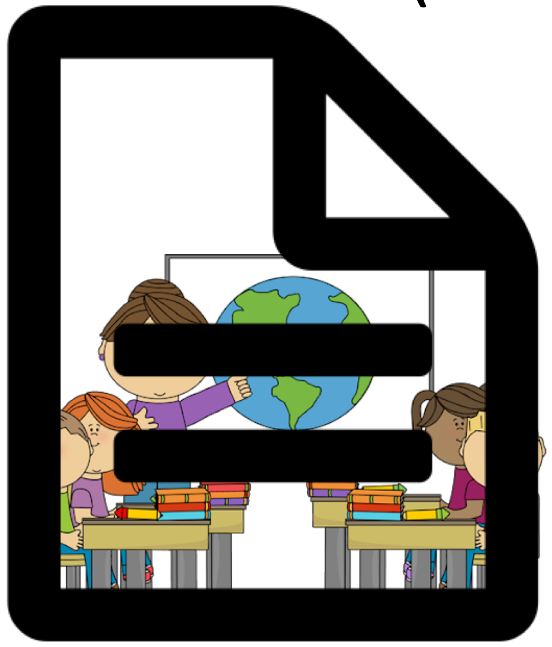
- Complexity grows very fast when solving real problems
- Every little feature on a web page or an app needs its own code and data and is related to everything else
- Solution:
  - Divide the software to modules that have defined interfaces to use
    - Interface: functions, data structures, network protocols
  - Use object oriented model to hide the actual data structures
  - Store modules in separate files and divide up the work among developers

Put Classes to their own File (Module)



Put Classes to their own File (Module)





In our Example:

Class = Student
Module = student
Filename (of the module with the class Student) = student.py

Implement your class once and use it everywhere!

A module is a file containing Python definitions and statements. The file name is the module name with the suffix ".py" appended.
Place your module files in the same directory as the main program.

Concept	Example
Module	student
Class	Student

Constructor initializer method	def _ _ init_ _ (self, myName)
	    self._ _name = myName

Initialization	Student1 = student.Student(“Barbara”, 123)
instance	student1 = student.Student(“Barbara”, 123) student1
attribute	_ _ name

Method	def add_course(self,course):
	        self._ _courses.append(course)

Method call	student1.add_course(“A”)

Function	def read_input():
	    return(input(“Enter your input\n”))

Function call	myInput = read_input()

Recap:

```
class Student:

    def _ _ init_ _(self, myName, myNumber):
        self._ _ name = myName
        self._ _id = myNumber
        self._ _grades = [ ]
        self._ _courses = [ ]

    def add_course(self,course):
        self._ _courses.append(course)

main():
    student1 = Student(”Barbara”,123)
    student2 = Student(”Angelina”,564)
    studentRegistry = (student1,student2,student3)
    name = read_input()
    student1.add_course(”Basics in Programming”)
    student2.add_course(”Algorithms and Datastructures”)
```

Module: student
File: student.py

```
class Student:

    def _ _ init_ _(self, myName, myNumber):
        self._ _ name = myName
        self._ _id = myNumber
        self._ _grades = [ ]
        self._ _courses = [ ]

    def add_course(self,course):
        self._ _courses.append(course)
```

Main program

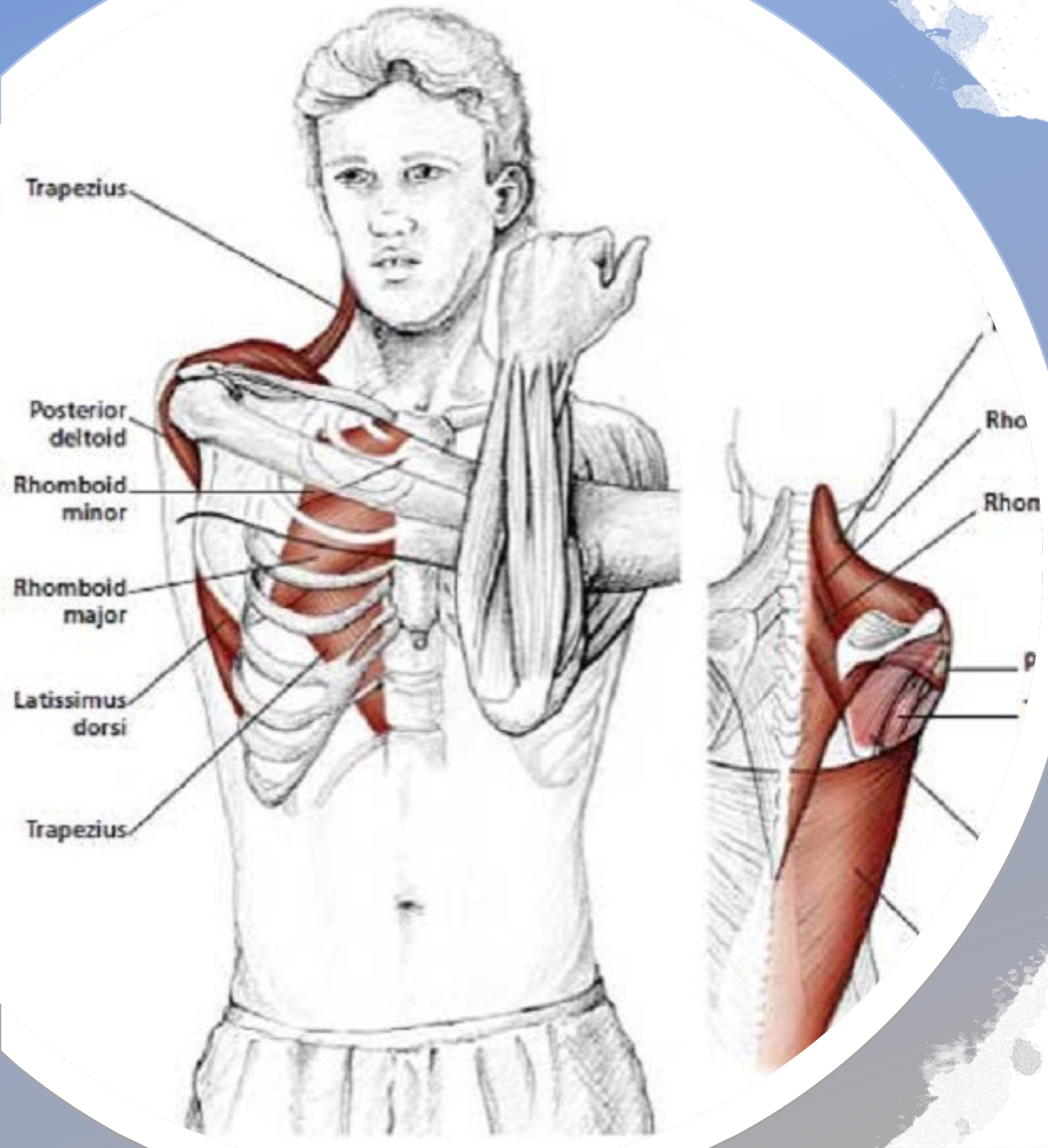
```
import student

    student1 = student.Student(”Barbara”,123)
    student2 = student.Student(”Angelina”,564)
    student1.add_course(”Basics in Programming”)
    student2.add_course(”Algorithms and Datastructures”)
```

Note the module name when creating an object, but not when using the object

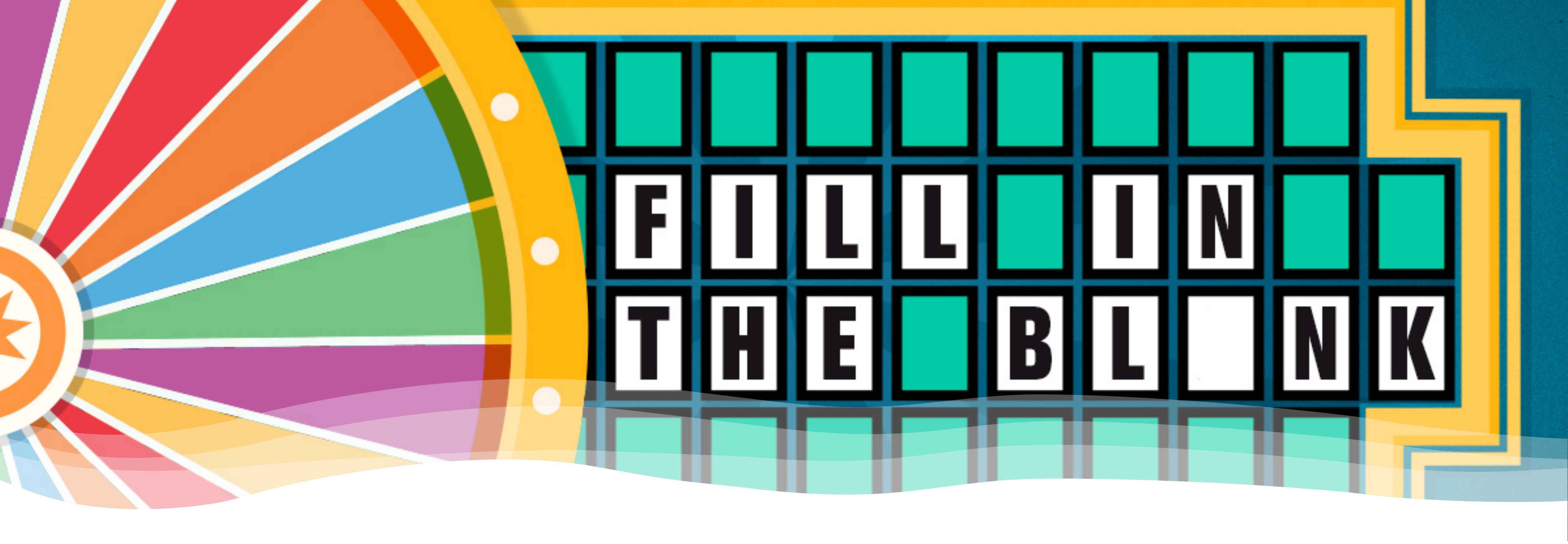
Think of class definition as a recipe and the object as the cake

Good to Know

- Guidelines:
  - Imports always at the start of the file
  - Import is only happening once per interpreter session,
    if you make changes to the module, you need to restart the interpreter
    (PyCharm does an automatic reload)
- There is much more, like packages → not part of this course
- The module does NOT need to be in the same directory, for our purpose it is easier to keep it that way



Break:
Move your Shoulders



Whats up with all the _ _ _ _ _ _ _ _ _ _ _

It is about who has access to what

_ _birthyear hides the attribute “birthyear” from the world outside of your class

student1._ _birthyear = 1999 is **not** valid in your main():

But Why?

It makes it way easier to structure and maintain your code. If there are changes, you only need to update the Class, not every program that uses it.

Example

```
Class Student1:

    def _ _init_ _(self, myName):
        self.name = myName
        self.age = 0


    def get_age(self):
        return self.age


main():

    myStudent.age = 15

    ........
    if myStudent.age < 18:

        print("sorry, you are underage")
```

```
Class Student2:

    def _ _init_ _(self, myName):
        self._ _name = myName
        self._ _age = 0

    def get_age(self):
        return self._ _ age

    def set_age(self, myAge):

        if (-0.75 <myAge<150):
            self._ _ age = myAge

main():

    mySTudent.set_age(15)

     ........
    age = myStudent.get_age()

    if age < 18:

        print("sorry, you are underage")
```

Example

```
Class Student1:

    def _ _init_ _(self, myName):
        self.name = myName
        self.age = 0

    def get_age(self):
        return self.age


main():

    myStudent.age = 15

    ........
    if myStudent.age < 18:

        print("sorry, you are underage")
```

Change attribute **age** to **birthyear**

```
Class Student2:

    def _ _init_ _(self, myName):
        self._ _name = myName
        self._ _age = 0

    def get_age(self):
        return self._ _ age

    def set_age(self, myAge):

        if (-0.75 <myAge<150):
            self._ _ age = myAge

main():

    myStudent.set_age(15)

     ........
    age = myStudent.get_age()

    if age < 18:

        print("sorry, you are underage")
```

Example

```
Class Student1:

   def _ _init_ _(self, myName):
      self.name = myName
      self.birthyear = 0

   def get_age(self):

      age = CUR_YEAR -self.birthyear
      return age


main():

   myStudent.age = 15

   ........
   if myStudent.age < 18:

      print("sorry, you are underage")
```

Change attribute **age** to **birthyear**

```
Class Student2:

   def _ _init_ _(self, myName):
      self._ _name = myName
      self._ _birthyear = 0

   def get_age(self):

      age = CUR_YEAR – self._ _birthyear
      return age

   def set_age(self, myAge):

      if (-0.75 <myAge<150):
         self._ _ birthyear = CUR_YEAR - myAge
main():

   myStudent.set_age(15)

   ........
   age = myStudent.get_age()

   if age < 18:

      print("sorry, you are underage")
```

Example

```
Class Student1:
    def _ _init_ _(self, myName):
        self.name = myName
        self.birthyear = 0
    def get_age(self):
        age = CUR_YEAR -self.birthyear
        return age


main():
    myStudent.age = 15
    ........
    if myStudent.age < 18:
        print("sorry, you are underage")
```

Change attribute **age** to **birthyear**

```
Class Student2:
    def _ _init_ _(self, myName):
        self._ _name = myName
        self._ _birthyear = 0
    def get_age(self):
        age = CUR_YEAR – self._ _birthyear
        return age
    def set_age(self, myAge):
        if (-0.75 <myAge<150):
            self._ _ birthyear = CUR_YEAR - myAge
main():
    myStudent.set_age(15)
    ........
    age = myStudent.get_age()
    if age < 18:
        print("sorry, you are underage")
```

get set.

Getters and Setters

Use for **every** attribute set- and get-methods!

```
def  set_age(self,myAge):
    self._ _age = myAge

def  get_age(self):
    return self._ _ age
```

Adding Attributes to a List

```
class Student

    def _ _init_ _(self, myName):
        self._ _name = myName
        self._ _grades = [ ]

    def add_grade(self,myGrade):
        if 0 <= myGrade <= 5:
            self._ _grades.add(myGrade)


main():
    student1 = Student("Barbara")

    student1.add_grade(5)
```

?

Adding Attributes to a List

```
class Student
    def _ _init_ _(self, myName):
        self._ _name = myName
        self._ _grades = [ ]

    def add_grade(self,myGrade):
        if 0 <= myGrade <= 5:
            self._ _grades.append(myGrade)
            return True
        else:
            return False

main():
    student1 = Student("Barbara")
    if student1.add_grade(5):
        print("grade added successfully")
    else:
        print("could not add grade to ", student1.get_name())
```

You like print()?

Do your own for your classes

_ _ methods _ _
cannot be called directly,
except by Python itself

```
class Student

   def _ _init_ _(self, myName, myNumber):

      self._ _name = myName
      self._ _ number = myNumber
      self._ _grades = [ ]


   def _ _str_ _(self):

      printString = "Student " + self._ _name + ", ID:" + self._ _nu
      return printString

main():
   student1 = Student("Barbara",123)
   print(student1)
```

```
Student Barbara, ID: 123
```

What is Supposed to be in a Class

_ _ init_ _ ← this is how you get an object of this class
_ _str_ _ ←to make life easier for others using your class

for all attributes (usually):
    set_attribute(attribute_value):
    get_attribute():

methods that are useful with your object / everyone needs with your object

    Eg, calculate average degree

Good to Know

- If you hide your attributes _ _
  - It is easier to update your class without updating other programs
  - It is cleaner
  - It is easier to ensure, that nothing fishy happens with your attribute (student.age = -5), as one can only set the age with the set_age method and you have control over that
  - It is not really true, that it cannot be accessed from outside your class, but it is not as easy
- Use separate set_attribute(value) and get_attribute() for all your attributes
- Use return True/False with setters

Objects in Lists

```
def main():
    studDirectory = ()
    newStudent = Student(”Visa”,568)
    studDirectory.append(newStudent)
    newStudent = Student(”Victoria”,784)
    studDirectory.append(newStudent)
    studDirectory.append(Student(”Taige”,778))
    thisName =studDirectory[1].get_name()

    for person in studDirectory:
        print(person)
```

Lists are a way to keep track of your objects

Why so Complicated?

- Why student1 = **student.**Student("Tim",1)?
- Why getters and setters?
- Why _ _variables?
- Why add_something has to return True or False?

- Reason: large systems, long life spans
- Code is written to be read and understood by other people
    - Need to maintain and update software
    - (Computer also reads code, but it does not need to understand)
- Modules and data hiding isolate components and allow re-use and independent maintenance

OO-programming and procedural programming can be mixed

```
def main():
    studDirectory = ()
    newStudent = Student(”Visa”,568)
    studDirectory.append(newStudent)
    newStudent = Student(”Victoria”,784)
    studDirectory.append(newStudent)
    studDirectory.append(Student(”Taige”,778))
    thisName =studDirectory[1].get_name()
    bestStudent = findBestInClass(studDirectory, ”Basics in Programming”)
    for person in studDirectory:
        print(person)
```

```
def findBestInClass(studDir,myClass):
    curbestGrad = 0
    for myStudent in studDir:
        if myClass in myStudent.get_courses():
            if myStudent.get_grade() > curbestGrad:
                curBestGrade = myStudent.get_grade()
                curBestStudent = myStudent
    return curBestStudent
```

So, What is the Object Oriented?

- A way to
  - think about the subjects of our programs
  - model reality with abstractions
  - separate tasks to manageable modules and re-use the modules for various needs
  - hide the details of implementation and provide specific services
  - allow improving different parts of software at the same time

"That's all Folks!"
Isberg©