Lecture 7
Object Oriented Programming

Jan Tekülve
jan.tekuelve@ini.rub.de

Computer Science and Mathematics
Preparatory Course

04.10.2018
Overview

1. Object Oriented Programming
   ▶ What is OOP?
   ▶ Classes vs. Instances
   ▶ Example Project
   ▶ Inheritance
   ▶ Modules in Python

2. Tasks

3. Outlook: Scientific Programming
   ▶ The Numpy Module
   ▶ Matrix Calculation with Numpy
Programming Paradigms

Procedural Programming

- A problem is solved by manipulating data structures through procedures
- The key is to write the right logic
- Efficiency is a main focus of procedural programming
Programming Paradigms

Procedural Programming

- A problem is solved by manipulating data structures through procedures
- The key is to write the right logic
- Efficiency is a main focus of procedural programming

Object oriented Programming

- A problem is solved by modeling it’s processes
- The key is to figure out the relevant entities and their relations
- Programming Logic is tightly coupled to entities
Classes vs. Objects

Class

Person
first name
last name
age
e-mail
Classes vs. Objects

Class

Person
  first name
  last name
  age
  email

Objects (Instances)

Alice
  Anderson
  28
  a.anders@gmail.com

Rob
  Robertson
  17
  cool_dude@aol.com
Classes Bind Variables Together

- Instead of writing something like this

```python
#Alice's attributes
aplace_name = "Alice"
aplace_last_name = "Anderson"
aplace_age = 28
```
Classes Bind Variables Together

▶ Instead of writing something like this

```python
#Alice’s attributes
alice_name = "Alice"
alice_last_name = "Anderson"
alice_age = 28
```

▶ Objects encapsulate multiple variables in one place

```python
#A Person-object variable
alice = Person("Alice", "Anderson", 28)
```
Classes are Advanced Data Types

- Object variables can be treated like simple types

  #Two Person-object variables
  alice = Person("Alice","Anderson",28)
  rob = Person("Rob","Robertson",17)
  #Objects can be stored in lists
  myPersonList = [] # I want to manage persons
  myPersonList.append(rob)
  #Objects can be arguments of self-defined functions
  calculate_year_of_birth(alice)
Class Definition

- A class needs to be defined

```python
class Person:  # This defines the class Name
    # The __init__ function is responsible for class creation and defines its’ attributes
    def __init__(self, first_name, last_name, age):
        # The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

- This is enough to create a class-object

```python
robbi = Person("Rob", "Robertson", 17)
```
Accessing Class Attributes

- Class attributes can be accessed via the `.` operator

```python
robbý = Person("Rob","Robertson",17)

f_name = robbý.first_name  # "Rob"
l_name = robbý.last_name   # "Robertson"
age = robbý.age           # 17
```
Accessing Class Attributes

- Class attributes can be accessed via the ‘.’ operator

```python
robbie = Person("Rob", "Robertson", 17)

f_name = robbie.first_name  # "Rob"
l_name = robbie.last_name  # "Robertson"
age = robbie.age  # 17
```

- They can also be assigned after initialization

```python
robbie.age = 18  # As he gets older
robbie.l_name = "Peterson"  # If he marries
```
Objects and Functions

- We can use objects as function arguments

```python
# Definition
def print_info(person):
    print(person.first_name + " " + person.last_name + " is " + str(person.age) + " years old.")
```

Usage:
```
robby = Person("Rob", "Robertson", 17)
print_info(robby)
# This prints: "Rob Robertson is 17 years old"

alice = Person("Alice", "Anderson", 28)
print_info(alice)
# This prints: "Alice Anderson is 28 years old"
```
Objects and Functions

- We can use objects as function arguments

```python
# Definition
def print_info(person):
    print(person.first_name + " " + person.last_name + " is " + str(person.age) + " years old."

# Usage:
robby = Person("Rob","Robertson",17)
print_info(robby)
# This prints: "Rob Robertson is 17 years old"

alice = Person("Alice","Anderson",28)
print_info(alice)
# This prints: "Alice Anderson is 28 years old"
```
Function Encapsulation

- Functions can even be defined inside classes

```python
class Person:  # This defines the class Name
    # The __init__ function
    def __init__(self, first_name, last_name, age):
        # The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    # Our print_info function
    def print_info(self):
        print(self.first_name + " " + self.last_name + " is " + str(self.age) + " years old.")
```
Function Encapsulation

- A function can be called directly from the object

```python
robbi = Person("Rob", "Robertson", 17)
robbi.print_info()
#This prints: "Rob Robertson is 17 years old"

alice = Person("Alice", "Anderson", 28)
alice.print_info()
#This prints: "Alice Anderson is 28 years old"
```
Function Encapsulation

- A function can be called directly from the object

```python
robbie = Person("Rob","Robertson",17)
robbie.print_info()
#This prints: "Rob Robertson is 17 years old"

alice = Person("Alice","Anderson",28)
alice.print_info()
#This prints: "Alice Anderson is 28 years old"
```

- This way a potential programmer/user does not need to know the internal structure of the particular class, e.g. `list.append()`.
Course Management Program

- We want to write a program for the university
- It should give an overview over the different courses
- It should track each course, its lecturer and its students
Course Management Program

- We want to write a program for the university
- It should give an overview over the different courses
- It should track each course, its lecturer and its students

How would an OOP model look like?
Course Management Program

Course

name
year
id_number
lecturer
student_list
Course Management Program

Course
- name
- year
- id_number
- lecturer
- student_list

Lecturer
- first name
- last name
- age
- email
- bank_account
Course Management Program

Course
- name
- year
- id_number
- lecturer
- student_list

Lecturer
- first name
- last name
- age
- email
- bank_account

Student
- first name
- last name
- age
- email
- student_id
- grade
Example Code

- The course class

```python
class Course:  #This defines the class Name
    #The __init__ function
    def __init__(self, name, year, id_number, lecturer):
        #The passed values are stored in the class
        self.name = name
        self.year = year
        self.id_number = id_number
        self.lecturer = lecturer
        self.student_list = []  #empty upon creation
```
Example Code

The lecturer class

class Lecturer: #This defines the class Name
    #The __init__ function
    def __init__(self, first_name, last_name, age, email, bank_account):
        #The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.email = email
        self.bank_account = bank_account
Example Code

Create the Course

```python
lecturer_jan = Lecturer("Jan","Tekuelve",29,"jan.tekuelve@uni.rub.de",1234567)
cscience_course = Course("Computer Science and Mathematics",2018,1234,lecturer_jan)
```

At the end of the year access the bank account:
```python
c_bank_account = cscience_course.lecturer.bank_account
```

This works independent of course and lecturer.
Example Code

Create the Course

```python
lecturer_jan = Lecturer("Jan","Tekuelle",29,"jan.
  tekuelle@ini.rub.de",1234567)
cscience_course = Course("Computer Science and
  Mathematics",2018,1234,lecturer_jan)
```

At the end of the year access the bank account:

```python
c_bank_account = cscience_course.lecturer.bank_account
```
Example Code

Create the Course

```python
lecturer_jan = Lecturer("Jan","Tekuelve",29,"jan.tekuelve@ini.rub.de",1234567)
cscience_course = Course("Computer Science and Mathematics",2018,1234,lecturer_jan)
```

At the end of the year access the bank account:

```python
c_bank_account = cscience_course.lecturer.bank_account
```

This works independent of course and lecturer
The Student Class

▶ This class looks similar to the lecturer

class Student:  #This defines the class Name
    #The __init__ function
    def __init__(self, first_name,last_name,age,email,student_id):
        #The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.email = email
        self.student_id = student_id
        self.grade = -1
Code Redundancy

Course
- name
- year
- id_number
- lecturer
- student_list

Lecturer
- first name
- last name
- age
- email
- bank_account

Student
- first name
- last name
- age
- email
- student_id
- grade
Code Redundancy

Course
name
year
id_number
lecturer
student_list

Lecturer
first name
last name
age
e-mail
bank_account

Student
first name
last name
age
e-mail
student_id
grade
Code Redundancy

Course
- name
- year
- id_number
- lecturer
- student_list

Lecturer
- first name
- last name
- age
- email
- bank_account

Person
- first name
- last name
- age
- email

Student
- first name
- last name
- age
- email
- student_id
- grade
Code Redundancy

- **Course**
  - name
  - year
  - id_number
  - lecturer
  - student_list

- **Lecturer**
  - first name
  - last name
  - age
  - email
  - bank_account

- **Person**
  - first name
  - last name
  - age
  - email

- **Student**
  - first name
  - last name
  - age
  - email
  - student_id
  - grade
Code Redundancy

Course
name
year
id_number
lecturer
student_list

Lecturer
bank_account

Person
first name
last name
age
email

Student
student_id
grade
The Person Class

We will use the Class Person as Super-Class

class Person:  #This defines the class Name
    #The __init__ function
    def __init__(self, first_name, last_name, age, email):
        #The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.email = email
Inheritance

- Lecturer and Student will inherit from Person

```python
class Lecturer(Person):  # Brackets declare inheritance
    # The __init__ function is overridden
    def __init__(self,f_name,l_name,age,email,b_acc):
        # The super() calls the parent function
        super().__init__(f_name,l_name,age,email)
        self.bank_account = b_acc

class Student(Person):  # Brackets declare inheritance
    # The __init__ function is overridden
    def __init__(self,f_name,l_name,age,email,stud_id):
        super().__init__(f_name,l_name,age,email)
        self.student_id = stud_id
        self.grade = -1
```
Modifying the Parent Class

- Functions of the parent class are available to child classes

```python
class Person:  #This defines the class Name
    def __init__(self, first_name, last_name, age, email):
        #The passed values are stored in the class
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.email = email

    #Our print_info function
    def print_info(self):  #Note how the argument changed
        print(self.first_name +" " +self.last_name +" is " +str(self.age) +" years old.")
```

Using Parent Functions

Functions of the parent class are available to child classes

```python
student_rob = Student("Rob","Robertson",25,"rob.
  robson@rub.de","108001024")
lecturer_jan = Lecturer("Jan","Tekuelve",29,"jan.
  tekuelve@ini.rub.de",1234567)

student_rob.print_info()
lecturer_jan.print_info()
#Prints:
#Rob Robertson is 25 years old.
#Jan Tekuelve is 29 years old.
```
Completing the Example

- The course needs to be able to add students

```python
#Inside the Course class
def enroll(self, student):
    self.student_list.append(student)
    #Enroll adds them to the course internal list
```

- Minimal example:

```python
cscience_course = Course("Computer Science and Mathematics", 2018, 1234, lecturer_jan)
student_rob = Student("Rob", "Robertson", 25, "rob.robson@rub.de", "108001024")
cscience_course.enroll(student_rob)
```
Creating your own Python Modules

- Class definitions can be stored in separate module
- E.g. if you save the above class definitions in a file unimanager.py
Creating your own Python Modules

- Class definitions can be stored in separate module

- E.g. if you save the above class definitions in a file `unimanager.py`

- You can access the definitions in another script from the same folder:

```python
import unimanager
student_rob = unimanager.Student("Rob","Robertson",25,"rob.robson@rub.de","108001024")
```
Creating your own Python Modules

- Class definitions can be stored in separate module
- E.g. if you save the above class definitions in a file `unimanager.py`
- You can access the definitions in another script from the same folder:
  ```python
  import unimanager
  student_rob = unimanager.Student("Rob","Robertson",25,"rob.robson@rub.de","108001024")
  ```
- This allows for flexible re-usability of code
Advantages/Disadvantages of OOP

Advantages:

- Design Benefit: Real/World processes are easily transferable in code
- Modularity: Extending and reusing software is easy
- Software Maintenance: Modular code is easier to debug
Advantages/Disadvantages of OOP

**Advantages:**

- Design Benefit: Real/World processes are easily transferable in code
- Modularity: Extending and reusing software is easy
- Software Maintenance: Modular code is easier to debug

**Disadvantages:**

- Design Overhead: Modeling requires longer initial development time
- Originally OOP required more “coding”
Tasks

1. Download today’s class definitions `unimanager.py` and create a separate script that uses this module to create a course, a lecturer and three sample students.
   - Enroll all students to the course.
   - After enrolling iterate through the student list to print the info of all enrolled students. You can access the student_list via the course object.
   - In the loop use the `print_info()` function.

2. Add a `print_info()` function to the class definition of Course in `unimanager.py`. This function should print the course name, its lecturer and each student of the course with his/her student ID.
   - The function should be defined in the Course class and its only argument should be self
   - The course name, the lecturer and its student_list can be accessed via the self keyword.
The Numpy Module

- Numpy is part of SciPy the module for scientific programming
- It should have been installed with matplotlib
- It is usually imported like this:

```python
import numpy as np
```
The Numpy Array

- Numpy brings its own data structure the numpy array

```python
import numpy as np

# Arrays can be created from lists
array_example = np.array([1, 6, 7, 9])

# Arrays can be created with arange
# An array with numbers from 4 to 5 and step size 0.2
array2 = np.arange(4, 5, 0.2)  # 5 is not in the array

print(array2)  # [4.0 4.2 4.4 4.6 4.8]
```

- Elements of an array can be manipulated simultaneously

```python
array3 = array2 * array2  # For example with multiplication

print(array3)  # [16.0 16.64 19.36 21.16 23.04]
```
Matplotlib and Numpy

Plotting $\sin(x)$ from 0 to $\pi$ with lists

```python
listX = []
listY = []
step_size = 0.5
for i in range(0, math.pi / step_size):
    xValue = i * step_size
    listX.append(xValue)
    listY.append(math.sin(xValue))
plt.plot(listX, listY)
```

Plotting $\sin(x)$ from 0 to $\pi$ with numpy

```python
xValues = np.arange(0, math.pi, 0.5)
yValues = np.sin(xValues)
plt.plot(xValues, yValues)
```
Numpy Arrays as Matrices

Creating the following matrix: \[ A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \]
**Numpy Arrays as Matrices**

- Creating the following matrix: \( \mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \)

- In numpy a matrix can be created from a multi-dimensional list

```python
#This creates a 3x4 Matrix
A = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```
Numpy Arrays as Matrices

- Creating the following matrix: \( \mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \)

- In numpy a matrix can be created from a multi-dimensional list

```python
# This creates a 3x4 Matrix
A = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

- Numpy treats such an array as a matrix

```python
arr_dim = A.shape # Gives you the shape of your matrix
print(arr_dim) # Prints (3,4)
# Access elements with indexing
single_number = A[1,3] # 8, 2nd list, 4th element
num2 = A[0,1] # 2, 1st list, 2nd element
```
Matrix Operations in Numpy

- Matrix Addition: 
  \[
  \begin{pmatrix}
  1 & 2 & 3 \\
  5 & 6 & 7
  \end{pmatrix}
  + 
  \begin{pmatrix}
  3 & 5 & 1 \\
  5 & -3 & 1
  \end{pmatrix}
  = 
  \begin{pmatrix}
  4 & 7 & 4 \\
  10 & 3 & 8
  \end{pmatrix}
  \]

- In numpy code:

```python
A = np.array([[1,2,3], [5,6,7]])
B = np.array([[3,5,1], [5,-3,1]])
C = A + B
D = A - B #Subtraction works analogously
print(D) #\begin{pmatrix}
-2 & -3 & 2 \\
0 & 9 & 6
\end{pmatrix}
```
Matrix Operations in Numpy

Matrix Multiplication: 
\[
\begin{pmatrix}
1 & 2 & 3 \\
5 & 6 & 7
\end{pmatrix}
\times
\begin{pmatrix}
3 & 5 \\
5 & -3 \\
1 & 1
\end{pmatrix}
= 
\begin{pmatrix}
16 & 2 \\
52 & 14
\end{pmatrix}
\]

In numpy code:

```python
A = np.array([[1,2,3], [5,6,7]])
E = np.array([[3,5], [5,-3],[1,1]])
F = np.matmul(A,E)
print(F) # [[16,2],[52,14]]
```

Do not confuse with element-wise multiplication:

```python
A = np.array([[1,2,3], [5,6,7]])
B = np.array([[3,5,1], [5,-3,1]])
G = A*B # [[3,10,3],[25,-18,7]]
```
Matrix Operations in Numpy

- **Matrix Multiplication:**
  
  $\begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{pmatrix} \times \begin{pmatrix} 3 & 5 \\ 5 & -3 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 16 & 2 \\ 52 & 14 \end{pmatrix}$

- **In numpy code:**

  ```python
  A = np.array([[1,2,3], [5,6,7]])
  E = np.array([[3,5], [5,-3],[1,1]])
  F = np.matmul(A,E)
  print(F) # [[16,2], [52,14]]
  ```

- **Do not confuse with element-wise multiplication**

  ```python
  A = np.array([[1,2,3], [5,6,7]])
  B = np.array([[3,5,1], [5,-3,1]])
  G = A*B # [[3,10,3], [25,-18,7]]
  ```
Matrix Operations in Numpy

- It also works for vectors:

\[
\langle v_1, v_2 \rangle = v_1^T v_2 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 5 \\ 1 \end{pmatrix} = 16
\]

- In numpy code:

```python
V1 = np.array([1,2,3])
V2 = np.array([3,5,1])
R = np.matmul(V1, V2)
print(R) # 16
```
Matrix Operations in Numpy

- It also works for vectors:

\[ \langle v_1, v_2 \rangle = v_1^T v_2 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 5 \\ 1 \end{pmatrix} = 16 \]

- In numpy code:

```python
V1 = np.array([1,2,3])
V2 = np.array([3,5,1])
R = np.matmul(V1,V2)
print(R) # 16
```

- Or vectors and matrices if you want to
Other helpful Operations

- Transpose Matrices: \( A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{pmatrix} \) \( A^T = \begin{pmatrix} 1 \\ 5 \\ 2 \\ 6 \\ 3 \\ 7 \end{pmatrix} \)

- In numpy:

\[
A = \text{np.array}([[[1, 2, 3], [5, 6, 7]]])
\]
\[
H = A.T \quad \# \quad [[1, 5], [2, 6], [3, 7]]
\]

- Element-wise summing across arrays:

\[
\text{sum} = \text{np.sum}(H) \quad \# 24,
\]
\[
V1 = \text{np.array}([1, 2, 3]) \quad \# \text{works also for 1D-arrays}
\]
\[
\text{sum}_v = \text{np.sum}(V1) \quad \# 6
\]