Part IV

Object-oriented Programming

What Is an Object?

- a car
- a cat
- a chair
- ...

- a molecule
- a star
- a galaxy
- ...

- anything that represents a real thing
- anything that represents an abstract thing
- anything which has some properties

How to Program Object-orientedly

- Change your way of thinking
- Change your point of view
- It is not just a technical question!

In Python?

- You do it all the time
- You just don’t know it
- In Python, everything is an object (event int)

```
a=4
a.__str__()
```

Theory: Classes and Objects

Classes

- A class is similiar to the definition of a type
- Specifies properties of the objects to be created
- Data to be stored once for all objects (class/static variable)
- Data to be stored in each object (member variables)
- Operations to be done with objects (methods)
- Represents a class of things/objects

Instances / Objects

- An instance/object is a concrete thing of the type of a class
- A class specifies properties, an object has specific values for the specified properties
- For each class, there can be an arbitrary number of objects
- Each object belongs exactly to one class (exception: polymorphism)

Theory: Constructor, Destructor

When creating an instance ...

- Memory is allocated
- A variable for storing the object is created
- Perhaps some things are initialized
- The initialisation is done with a constructor (`__init__(self,...)`)

When a variable is not used any more ...

- Memory has to be freed
- Perhaps some things have to be cleaned up
- The destructor (`__del__(self,...)`) is called
- BUT: No guarantee at which time it is called

```
class Example:
  def __init__(self, ...):
    do_something
  def __del__(self, ...):
    do_something
```

First Class Example

```
#!/usr/bin/python
class MPEEmployee(object):
  def __init__(self, name):
    self.name = name
  def printInfo(self):
    print("Name␣of␣this␣employee:␣", self.name)

person1 = MPEEmployee("John")
person2 = MPEEmployee("Mary")
person1.printInfo()
```

Theory: self

Usage of an object

- Access to methods: `object.method(...)`
- Example: `list.append('x')`
- Access to variables: `object.variable`
- Example: `complex.real`

Access within methods

- Concrete object is created outside the class definition

$\Rightarrow$ Methods of a class do not know the object name

$\Rightarrow$ All methods get additional formal parameter `self`

- `self` always is the first formal parameter, but is skipped at a call (actual parameter)
- Within methods: access to other methods and variables via `self.variable` and `self.method()`, resp.

Theory: Methods

- . . . implement the behaviour of objects.
- . . . represent all types of changes of an object after initialisation.
- accessor methods: give info on state of an object
- mutator methods: change state of an object

$\Rightarrow$ change (at least) one object/member variable

Procedural Approach to Operate an Oven

```
#!/usr/bin/python
def doBaking(temperature, mode):
    do_something

oventemperature = 180
ovenmode = 2
doBaking(oventemperature, ovenmode)
```

If i've got a whole bakery?

```
oventemperatures = []
ovenmodes = []
oventemperatures.append(190)
ovenmodes.append(2)
...
for i in range(len(oventemperatures)):
  do_baking(oventemperatures[i], ovenmodes[i])
```

Object Oriented Approach to Operate an Oven

```
#!/usr/bin/python
class Oven(object):
  def __init__(self, temperature, mode):
    self.temperature=temperature
    self.mode=mode
  def doBaking(self):
    do_something
myOven = Oven(180,2)
myOven.doBaking()
```

Again a whole bakery

```
ovens = []
ovens.append(Oven(190,2))
...
for oven in ovens:
  oven.doBaking()
```

What Are the Differences?

Procedural approach

- Procedures specify how to ”produce” something
- When implementing them, you have to think of actions
- You have to care about storing the data yourself

Object oriented approach

- Objects specify ”things” (real or abstract)
- When implementing them, you have to think of the thing and it’s properties
- All data is stored in the object itself (put corresponding data and code together!)
- Whenever lots of data elements are associated with an action, try to think object oriented

Static Member Variables & Methods

```
class MPEEmployee(object):
  num_employees = 0
  def __init__(self, name):
    self.name = name
    MPEEmployee.num_employees += 1
  def printInfo(self):
    print("Name␣of␣this␣employee:␣", self.name)
  @staticmethod
  def numPeople():
    return MPEEmployee.num_employees

person1 = MPEEmployee("John")
person2 = MPEEmployee("Mary")
person1.printInfo()
print(MPEEmployee.numPeople())
```

- **static member** variables: stored once for all objects of a class
- **static methods**: access static members without object instance

Inheritance

- Classes can inherit from a *base class / superclass*
- This makes them a *derived class / subclass* of the superclass
- All classes can be specialized ⇒ they become superclasses
- All classes together form a class hierarchy (taxonomy)
- Properties of the superclass can be taken over or can be redefined

```
#!/usr/bin/python
class PhDStudent(MPEEmployee):
  def __init__(self, name, thesis):
    self.thesis = thesis
    MPEEmployee.__init__(self, name)
  def printInfo(self):
    MPEEmployee.printInfo(self)
    print("%s works on %s" % (self.name, self.thesis))

person3 = PhDStudent("Terry", "Terry's thesis topic")
person3.printInfo()
```

Inheritance in Real Life

Information Hiding

public

- Per default: everything (all variables/methods)
- Comfortable (access from outside)
- Often dangerous / undesired

private

- External access (i.e. from outside the class) forbidden

$\Rightarrow$ Variables/methods not visible externally

$\Rightarrow$ Variables/methods not usable externally

- Syntax: starting with `__`, but not ending with `__` (`__privateVar`, `__privateMet()`)
- Attention: indirect access possible (`._class__privateVar`)

Special Variables and methods

- Variables and Methods starting with __, but not ending with __ like `__privateVar` and `__privateMet()` are private
- `__get__(i)`: Allows reading via `[]`
- `__set__(i, value)`: Allows writing via `[]`
- `__add__(value)`: Defines an addition with `+`
- `__str__()`: used if object is a parameter for `print`

```
class Test:
  def __init__(self, val):
    self.value = val
  def __add__(self, b):
    return self.value + b.value
  def __mul__(self,b):
    return self.value * b.value
```