

Algorithms and Programming

Lecture 6 – Classes

Camelia Chira

Course content

- Introduction in the software development process
- Procedural programming
- Modular programming
- Abstract data types
- Software development principles
- Testing and debugging
- Recursion
- Complexity of algorithms
- Search and sorting algorithms
- Backtracking
- Recap

Last time

Abstract Data Types

- Classes
 - Flower example
 - Rational example

Today

- More on ADT
 - Data abstraction
 - Information hiding
 - Class attributes vs Instance attributes
 - Static methods

Unified Modelling Language (UML)

Classes

- Creating a new class:
 - Creates a new type of an object
 - Allows *instances* of that type
- A class instance can have:
 - Attributes (to maintain its state)
 - Methods (to modify its state)
- Class name is the type e.g. class Flower:
- Instance is one specific object e.g. f1 = Flower("rose", 5)
- A class introduces a new namespace

Data Abstraction

- Encapsulation
 - bundling of data with the methods that operate on that data

Data Abstraction

Data Encapsulation

Information Hiding

- Information hiding
 - the principle that some internal information or data is "hidden" so that it can not be changed by accident

Data Abstraction = Data Encapsulation + Data Hiding

Encapsulation

- Often accomplished by providing two types of methods
 - Getter methods
 - Setter methods

Obs. This does not mean that data attributes can be accessed only via the getter methods

```
class Flower:
   def __init__(self, n = "", p = 0):
        self.name = n
        self.price = p
   def getName(self):
            getter method: return the name of a flower
        return self.name
   def getPrice(self):
            getter method: return the price of a flower
        return self.price
   def setName(self, n):
            setter method: set the name of a flower
        111
        self.name = n
   def setPrice(self, p):
            setter method: set the price of a flower
        111
        self.price = p
```

Information hiding

- The internal representation of an object
 - Needs to be hidden outside the object's denition
 - Protect object integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state
- Python not great at information hiding
 - You can access data from outside class definition

```
print(f1.name)
```

• You can write data from outside class definition

```
f1.name = "Lily"
```

You can create data attributes for an instance from outside class definition

```
f1.colour = "Purple"
```

Not a good style to do any of these

Information hiding

- Divide the code into a public interface and a private implementation of that interface
- Data hiding in Python: public and private members
 - Data hiding in Python is based upon convention
 - Use the convention: _name or __name for fields, methods that are "private"
 - A name prefixed with an underscore (e.g. _spam) should be treated as non-public part of the API (should be considered an implementation detail and subject to change without notice)
 - A name prefixed with two underscores (e.g. __spam) is private and name mangling is employed (Python runtime)

Attribute types

- Private attributes
 - name
 - should only be used inside the class definition
- Protected (restricted) attributes
 - name
 - may be used but only under certain conditions
- Public attributes
 - name
 - can be freely used inside or outside class definition

Attribute types: example

```
class Flower:
    def __init__(self):
        self.name = "Lily"
        self._colour = "Purple"
        self._price = 10
```

```
>>> f = Flower()
>>> f.name
'Lily'
>>> f. colour
'Purple'
>>> f. price
Traceback (most recent call last):
 File "<pyshell#55>", line 1, in <module>
   f. price
AttributeError: 'Flower' object has no attribute ' price'
>>>
                     Information
                        hiding
```

Data encapsulation: revisited in the Flower example

```
class Flower:
   a flower is a structure of two elements: name (a string) and price (an integer)
   def __init__(self, n = "", p = 0):
       self. name = n
       self. price = p
   def getName(self):
       return self.__name
   def getPrice(self):
       return self. price
   def setName(self, n):
       self. name = n
   def setPrice(self, p):
       self. price = p
```

Class attributes vs. Instance attributes

- Instance attributes
 - Owned by the specific instances of the class
 - Usually different for each instance

```
f1 = Flower("rose", 5)
f2 = Flower("tulip", 3)
```

- Class attributes
 - Owned by the class itself
 - Same for all instances

```
>>> f1 = Flower()
>>> f1.counter
1
>>> Flower.counter
1
>>> f2 = Flower()
>>> Flower.counter
2
>>> f2.counter
2
>>> f1.counter
2
>>> f1.counter
```

Static methods

Class attributes can be private

```
class Flower:
   a flower is a structure of two elements: name (a string) and price (an integer)
   counter = 0
   def __init__(self, n = "", p = 0): | >>> f1.getCounter()
                                       >>> f2.getCounter()
           creates a new instance of F
                                       >>> Flower.getCounter()
       self. name = n
                                       Traceback (most recent call last):
       self. price = p
                                         File "<pyshell#74>", line 1, in <module>
       type(self).__counter += 1
                                           Flower.getCounter()
                                       TypeError: getCounter() missing 1 required positional argument: 'self'
   def getCounter(self):
                                       >>>
       return self. counter
```

Good idea?

Static methods

```
class Flower:
   a flower is a structure of two elements: name (a string) and price (an integer)
   counter = 0
   def __init__(self, n = "", p = 0):
           creates a new instance of Flower
       111
                                   >>> f1 = Flower()
       self. name = n
                                   >>> f2 = Flower()
       self.__price = p
                                   >>> Flower.getCounter()
       type(self).__counter += 1
                                   >>> f1.getCounter()
   def getCounter():
                                   Traceback (most recent call last):
       return Flower. counter
                                     File "<pyshell#80>", line 1, in <module>
                                       f1.getCounter()
                                   TypeError: getCounter() takes 0 positional arguments but 1 was given
                                   >>>
```

Static methods

- Add a line "@staticmethod" before method definition
- Use decorator syntax
- Do not require the self argument

```
class Flower:
   a flower is a structure of two elements: name (a string) and price (an integer)
   counter = 0
                                                                     >>> f1 = Flower()
                                                                     >>> f2 = Flower()
   def __init__(self, n = "", p = 0):
                                                                     >>> Flower.getCounter()
           creates a new instance of Flower
                                                                     >>> f1.getCounter()
       self.__name = n
       self.__price = p
                                                                     >>> f2.getCounter()
       type(self).__counter += 1
   @staticmethod
   def getCounter():
                                                                                                 16
       return Flower. counter
```

Example 1

```
class Student:
    __studentCount = 0
   def __init__(self, name=""):
        self.__name = name
        Student.__studentCount += 1
   def setName(self, name):
        self.__name = name
   def getName(self):
        return self.__name
   @staticmethod
   def getStudentCount():
       return Student. studentCount
```

```
s1 = Student()
s2 = Student()

s1.setName("Erin")
s2.setName("Carla")

print(s1.getName())
print(s2.getName())

print(Student.getStudentCount())
```

17

Example 2

```
class Account(object):
   num accounts = 0
   def __init__(self, name, balance):
       self.name = name
        self.balance = balance
        Account.num accounts += 1
   def del account(self):
        Account.num accounts -= 1
   def deposit(self, amt):
        self.balance = self.balance + amt
   def withdraw(self, amt):
        self.balance = self.balance - amt
   def inquiry(self):
        return self.balance
   @staticmethod
   def type():
        return "Current Account"
```

```
>>> a = Account("a1", 10)
>>> a.deposit
<bound method Account.deposit of < main .Account object at 0x02D5CFF0>>
>>> a.tvpe
<function Account.type at 0x02E34D68>
>>> a.tvpe()
'Current Account'
>>> a.deposit(30)
>>> a.inquiry()
40
>>> a.tvpe()
'Current Account'
>>> Account.type()
'Current Account'
>>> Account.num accounts()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    Account.num accounts()
TypeError: 'int' object is not callable
>>> Account.num accounts
>>> b = Account("a2", 20)
>>> b.num accounts
>>> Account.num accounts
>>> b.tvpe()
'Current Account'
```

Recap: Data Abstraction

- Encapsulation
 - bundling of data with the methods that operate on that data

Data Abstraction

Data Encapsulation

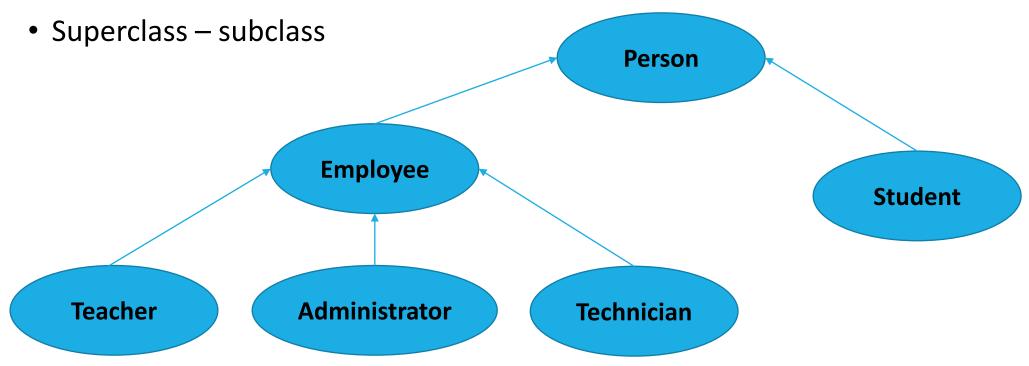
Information Hiding

- Information hiding
 - the principle that some internal information or data is "hidden" so that it can not be changed by accident

Data Abstraction = Data Encapsulation + Data Hiding

Inheritance

- Classes can inherit from other classes
 - Attributes and behaviour methods



Inheritance

```
class Person:
   def init (self, first, last):
      self.firstname = first
                                                  >>> x = Person("Bart", "Simpson")
      self.lastname = last
                                                  >>> y = Employee("Homer", "Simpson", "231")
                                                  >>> x.getFullName()
   def getFullName(self):
                                                  'Bart Simpson'
      return self.firstname + " " + self.lastname
                                                  >>> v.getFullName()
                                                  'Homer Simpson'
class Employee(Person):
                                                  >>> y.getEmployeeName()
   def init (self, first, last, staffid):
                                                  'Homer Simpson, 231'
      Person. init (self, first, last)
      self.staffnumber = staffid
   def getEmployeeName(self):
```

super().__init__(self, first, last)

Unified Modelling Language (UML)

• UML

- Standardized general-purpose modeling language
- Includes graphical notations to model concepts in the field of object-oriented software engineering
- Visual models of object-oriented applications

- Class diagram
 - Describe the structure of the application using
 - Classes (attributes and methods)
 - Relationships between classes

UML Class Diagram

Specification of a class

Flower

+name: String +price: Integer

+__init__()

+getName(): String

+setName(String)

+getPrice(): Integer

+setPrice(Integer)

+compare(Flower): Boolean

Visibility

- + -> public
- - -> private

Class name

Data section:

- Visibility
- Name
- Type

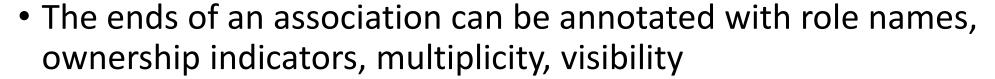
Method section:

- Visibility
- Name
- Arguments
- Type

```
class Flower:
    def init (self):
        self.name = ""
        self.price = ""
    def getName(self):
        return self.name
    def setName(self, n):
        self.name = n
    def getPrice(self):
        return self.price
    def setPrice(self, p):
        self. price = p
    def compare(self, other):
        if ((self.name == other.name) and
            (self.price == other.price)):
            return True
        else:
            return False
```

Relationships between classes: Association

- Association
 - Class A uses class B
 - Objects of A are connected to objects of B
- An association can be named



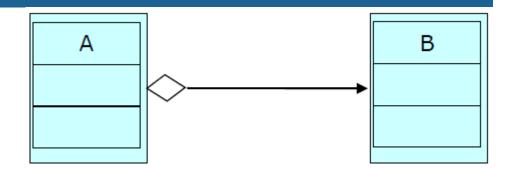
Association can be bi-directional as well as uni-directional



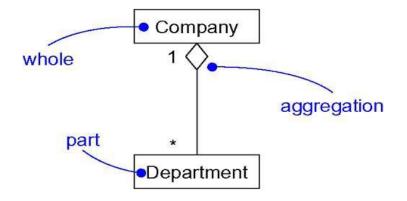


Relationships between classes: Aggregation

- Aggregation
 - A contains 1 or more B
 - B exists without A

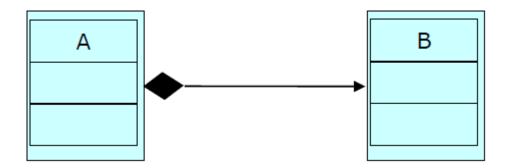


Special kind of association used to model a "whole/part" relationship



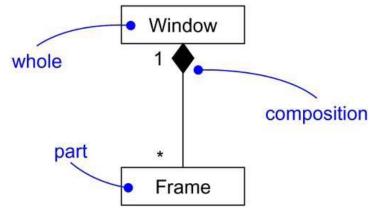
Relationships between classes: Composition

- Composition
 - A contains 1 or more B
 - B is created by A



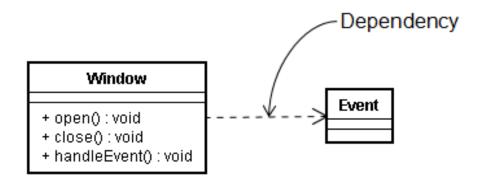
• Variation of simple aggregation: introduces a strong ownership and

coincident lifetime as part of the whole



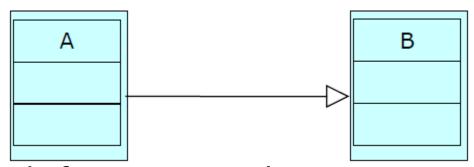
Relationships between classes: Dependency

- Dependency
 - A depends on B
- Shows that:
 - one class uses operations from another class, or
 - it uses variables or arguments typed by the other class
 - if the used class changes => the operation of the other class may be affected

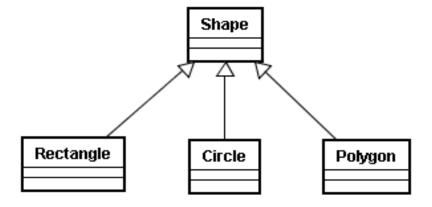


Relationships between classes: Generalization

- Generalization
 - A is a B



Child class inherits attributes and methods from parent class



Recap today

ADT

- Data encapsulation
- Information hiding
- Class attributes
- Instance attributes
- Static methods

• UML

- Class diagram
- Relationships between classes

Next time

Testing and debugging

Reading materials and useful links

- 1. The Python Programming Language https://www.python.org/
- 2. The Python Standard Library https://docs.python.org/3/library/index.html
- 3. The Python Tutorial https://docs.python.org/3/tutorial/
- 4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
- MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
- K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
- 7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html