BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 6 – Classes, UML, NumPy

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

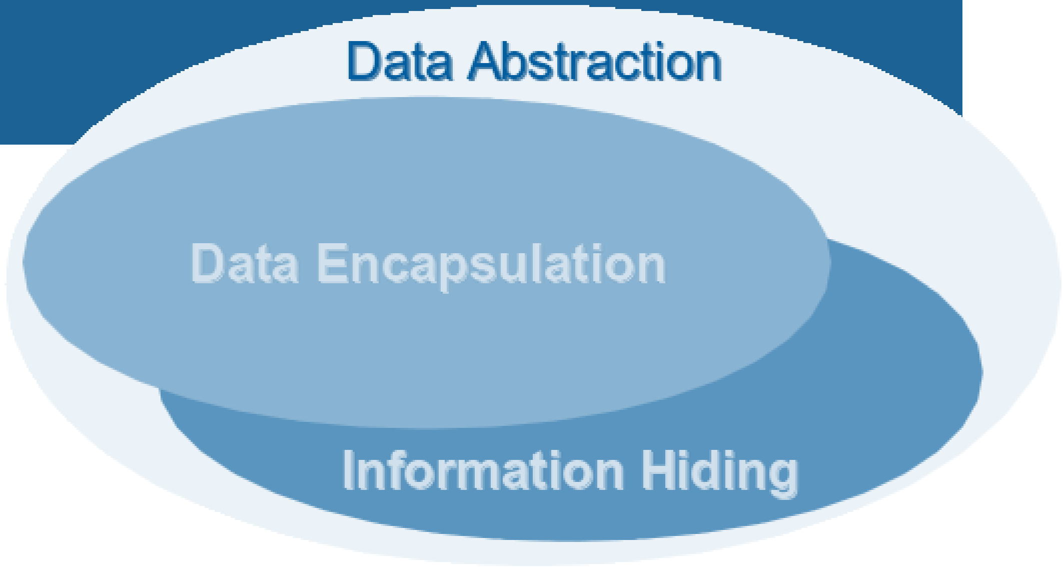
Today

- More on ADT
  - Data abstraction
  - Information hiding
  - Class attributes vs Instance attributes
  - Static methods
- Unified Modelling Language (UML)
- Special libraries: Numpy

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

- 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

```python
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
        '''
        self.name = n

    def setPrice(self, p):
        '''
            setter method: set the price of a flower
        '''
        self.price = p
```

Information hiding

- The internal representation of an object
  - Needs to be hidden outside the object's definition
  - 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

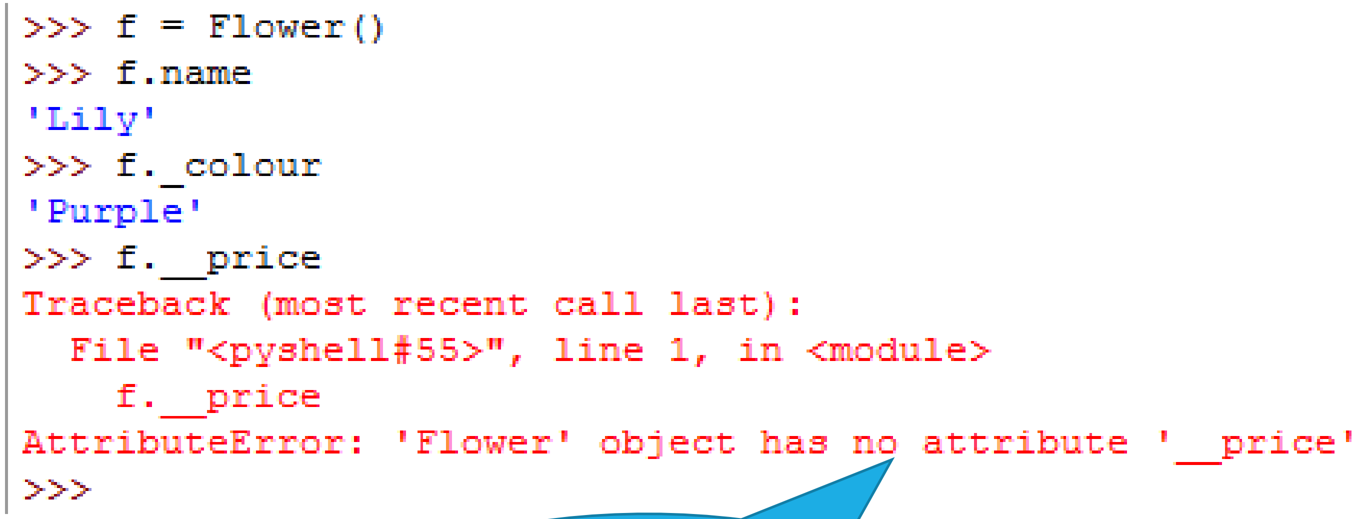
- Private attributes
  - __name
  - should only be used inside the class definition

- 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

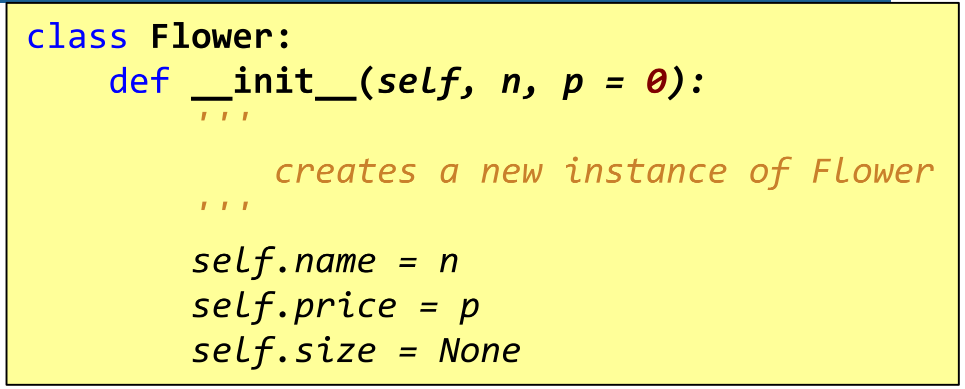
Data encapsulation: revisited in the Flower example

```
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

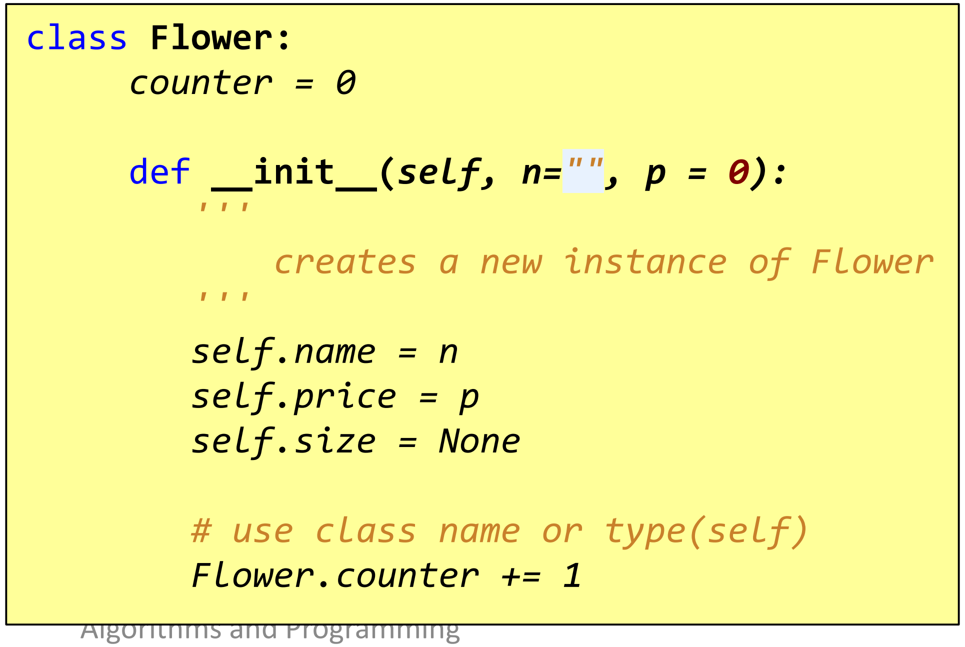
- Instance attributes
  - Owned by the specific instances of the class
  - Usually different for each instance

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

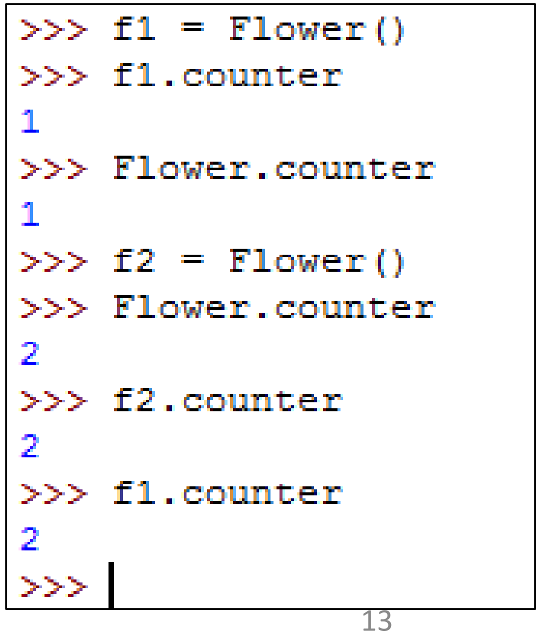


```
class Flower:
    def __init__(self, n, p = 0):
        '''
            creates a new instance of Flower
        '''
        self.name = n
        self.price = p
        self.size = None
```

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



```
class Flower:
    counter = 0

    def __init__(self, n="", p = 0):
        '''
            creates a new instance of Flower
        '''
        self.name = n
        self.price = p
        self.size = None

        # use class name or type(self)
        Flower.counter += 1
```

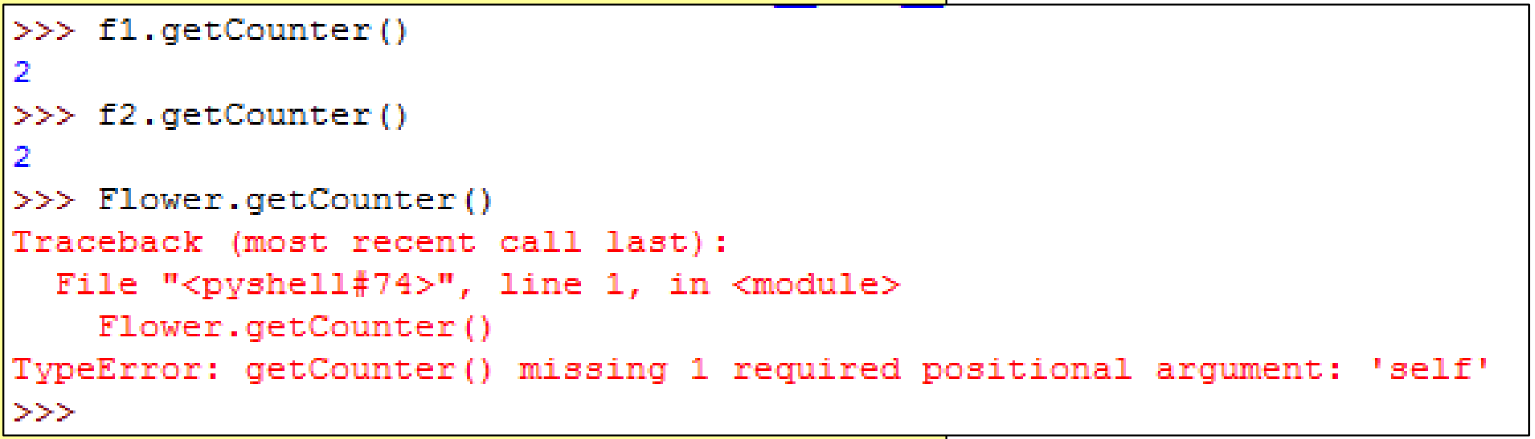


```
>>> f1 = Flower()
>>> f1.counter
1
>>> Flower.counter
1
>>> f2 = Flower()
>>> Flower.counter
2
>>> f2.counter
2
>>> f1.counter
2
>>> |
```

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):
        '''
            creates a new instance of F
        '''
        self.__name = n
        self.__price = p
        type(self).__counter += 1

    def getCounter(self):
        return self.__counter
```

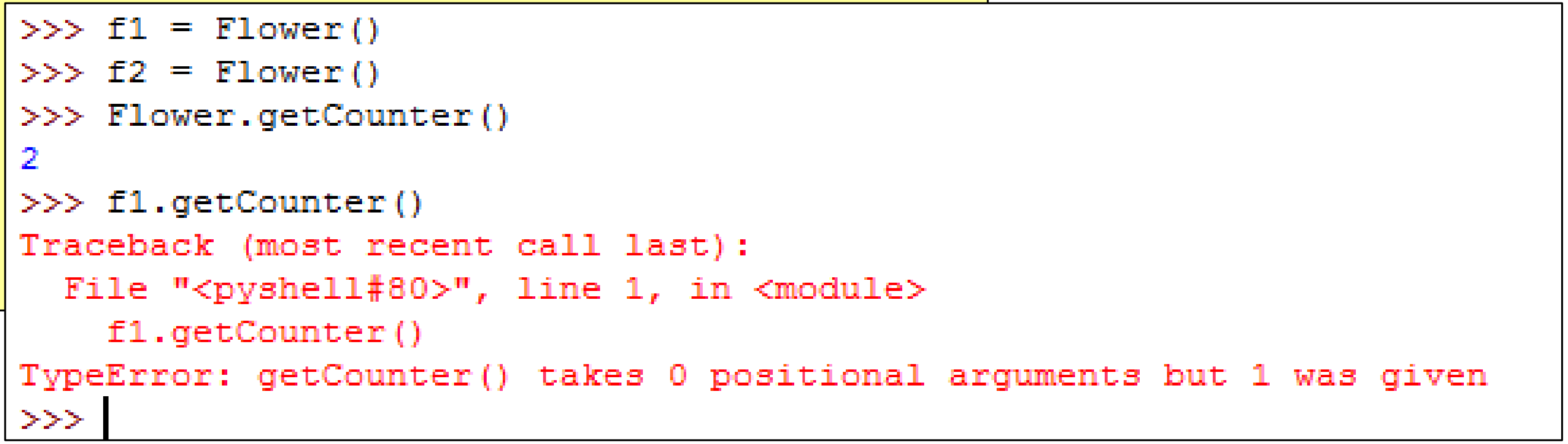


```
>>> f1.getCounter()
2
>>> f2.getCounter()
2
>>> Flower.getCounter()
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    Flower.getCounter()
TypeError: getCounter() missing 1 required positional argument: 'self'
>>>
```

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
        '''
        self.__name = n
        self.__price = p
        type(self).__counter += 1

    def getCounter():
        return Flower.__counter
```

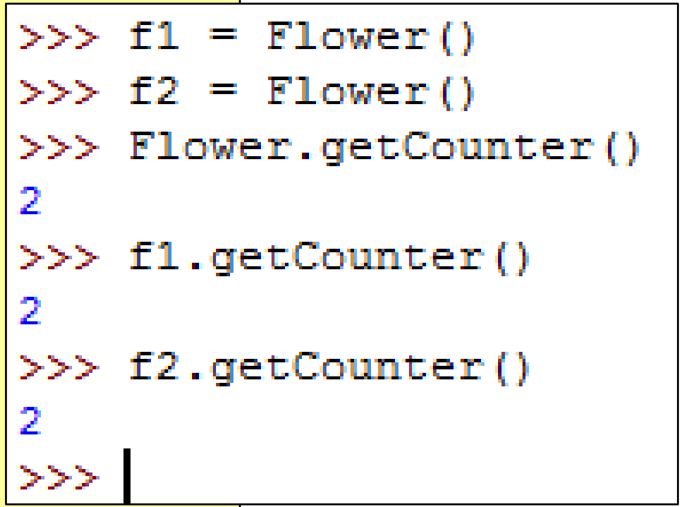


```
>>> f1 = Flower()
>>> f2 = Flower()
>>> Flower.getCounter()
2
>>> f1.getCounter()
Traceback (most recent call last):
  File "<pyshell#80>", line 1, in <module>
    f1.getCounter()
TypeError: getCounter() takes 0 positional arguments but 1 was given
>>> 
```

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

    def __init__(self, n = "", p = 0):
        '''
            creates a new instance of Flower
        '''
        self.__name = n
        self.__price = p
        type(self).__counter += 1

    @staticmethod
    def getCounter():
        return Flower.__counter
```



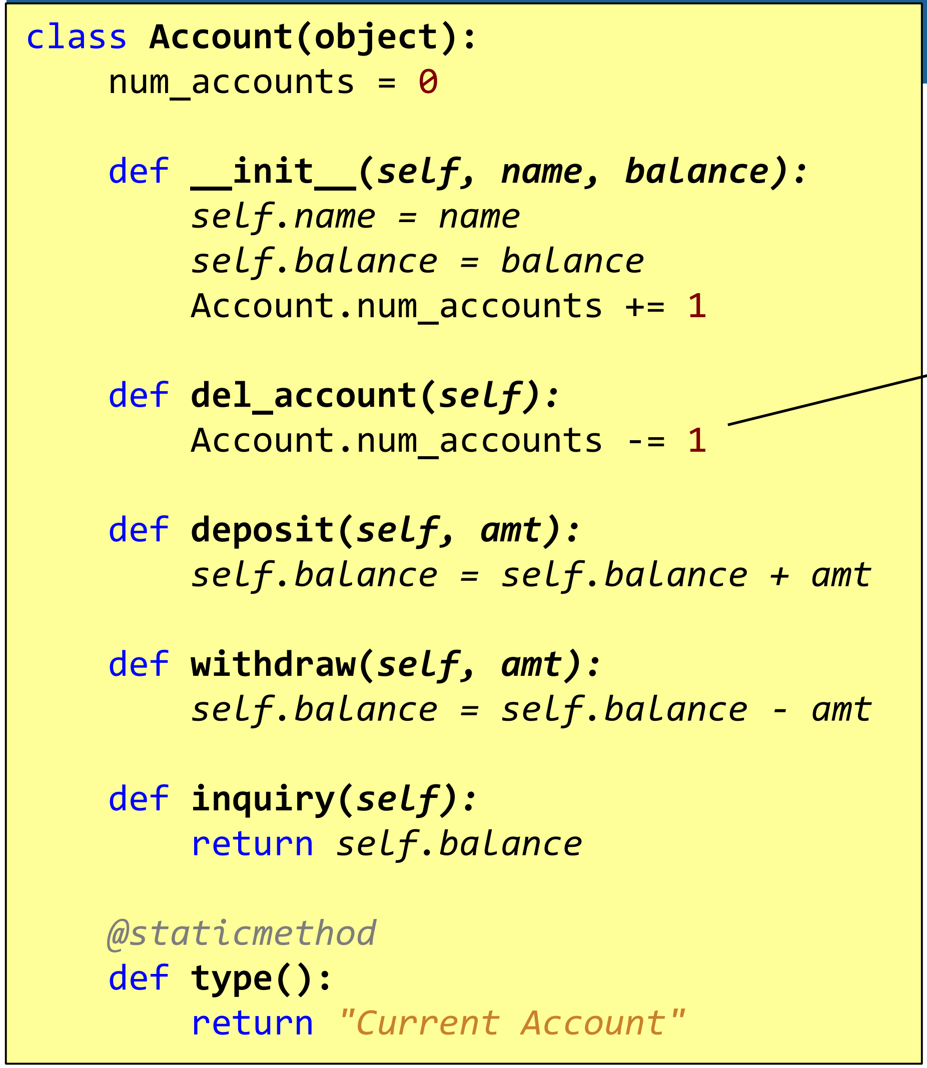
```
>>> f1 = Flower()
>>> f2 = Flower()
>>> Flower.getCounter()
2
>>> f1.getCounter()
2
>>> f2.getCounter()
2
>>> 
```

Example 1

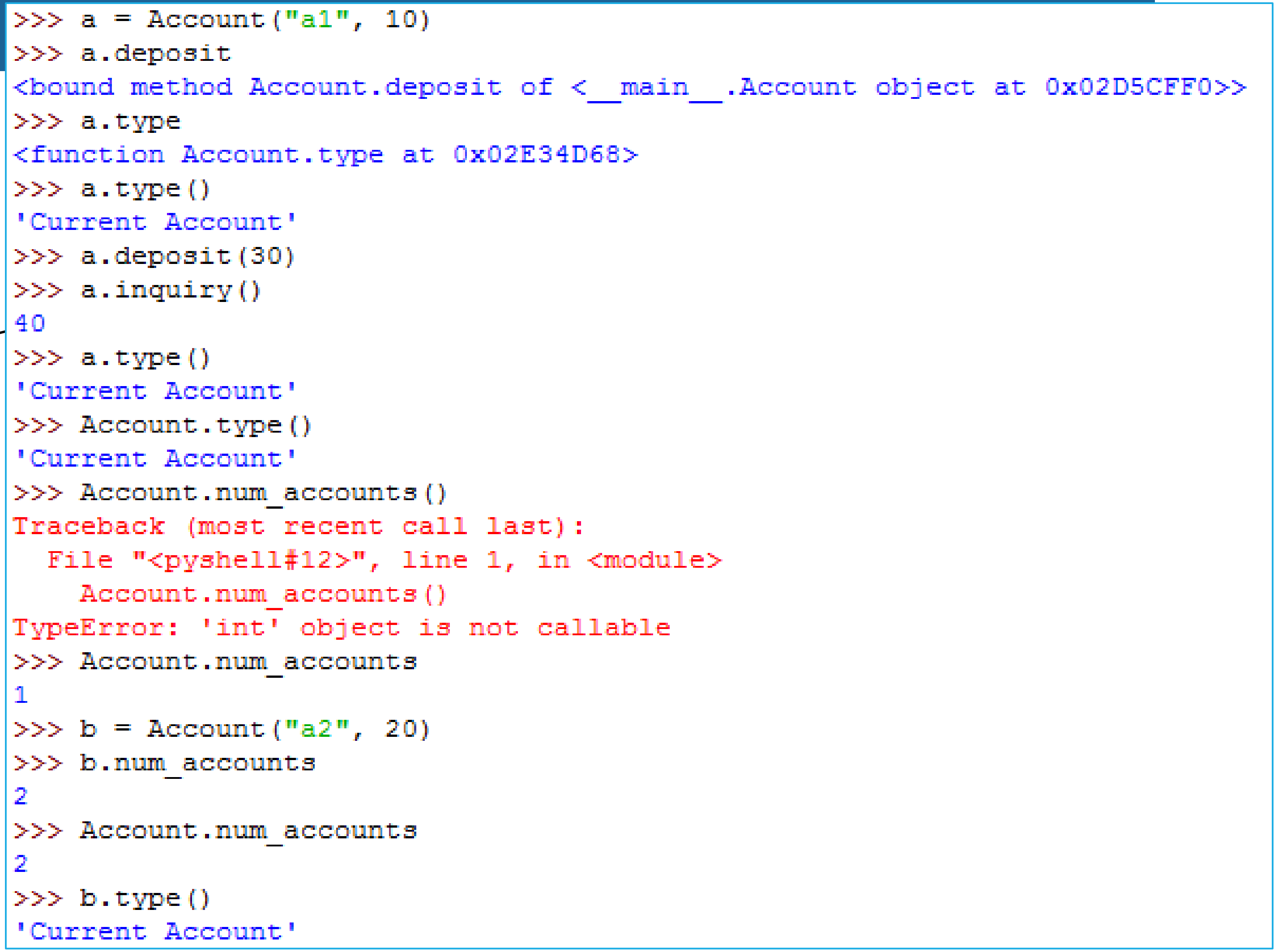
```
class Student:

    __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())
```

Example 2

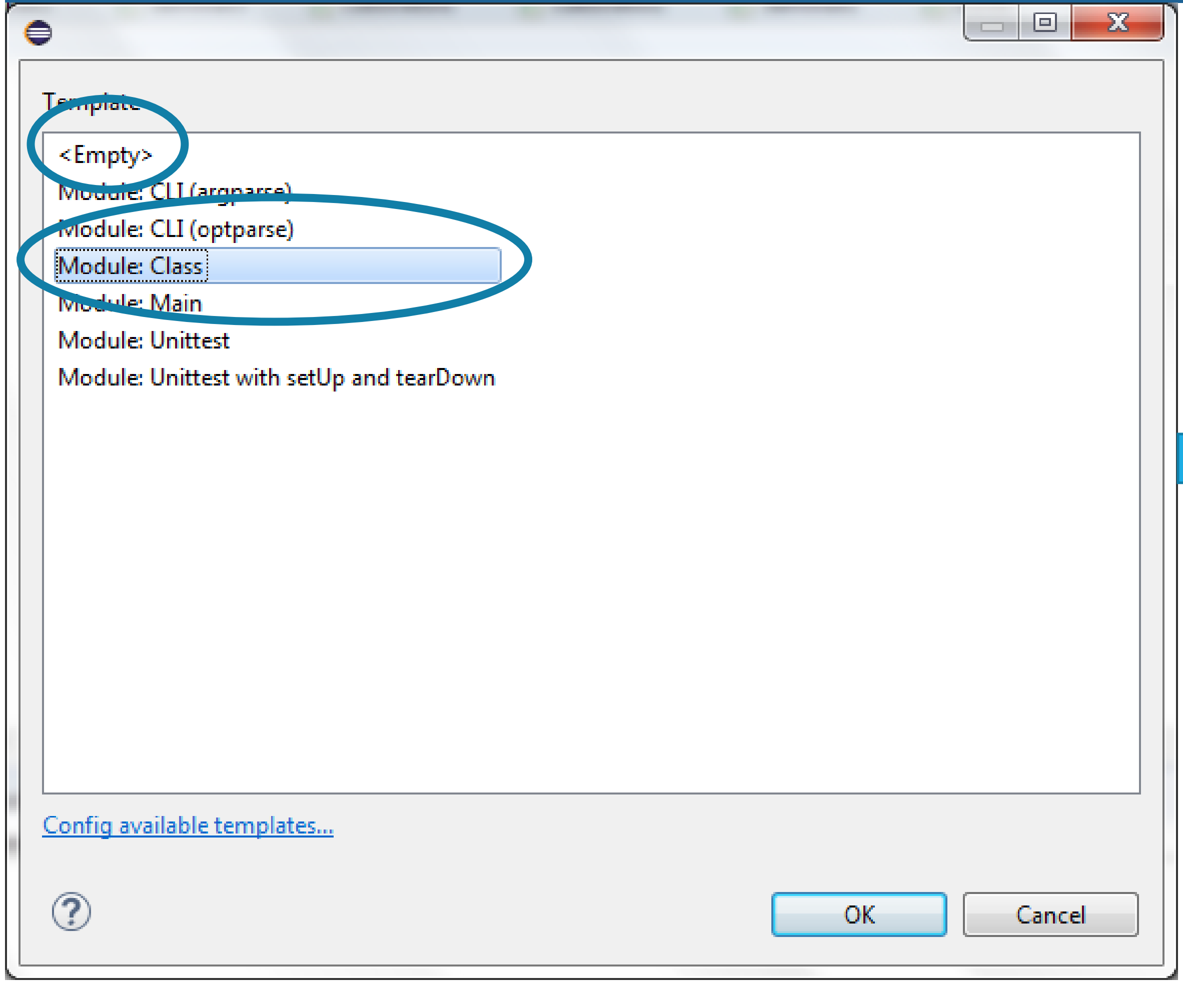


```
class Account(object):
    num_accounts = 0

    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.type
<function Account.type at 0x02E34D68>
>>> a.type()
'Current Account'
>>> a.deposit(30)
>>> a.inquiry()
40
>>> a.type()
'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
1
>>> b = Account("a2", 20)
>>> b.num_accounts
2
>>> Account.num_accounts
2
>>> b.type()
'Current Account'
```

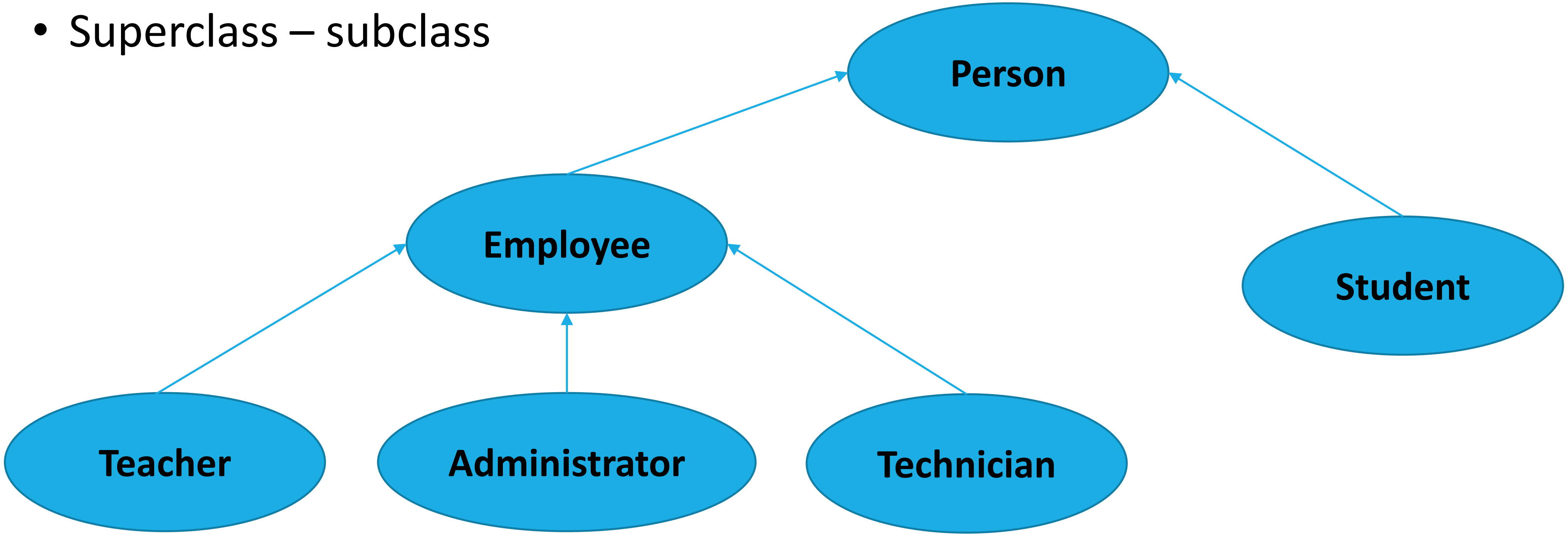
Creating a Python class in Eclipse



```
'''
Created on 11 Nov 2017

@author: cami
'''

class MyClass(object):
    '''
    classdocs
    '''


    def __init__(self, params):
        '''
        Constructor
        '''
```

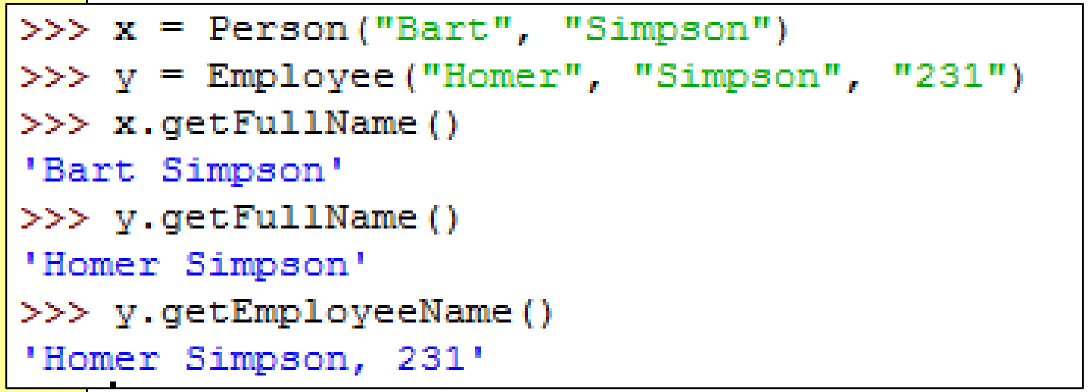
Inheritance

- Classes can inherit from other classes
  - Attributes and behaviour methods
  - Superclass – subclass



Inheritance

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def getFullName(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffid):
        Person.__init__(self, first, last)
        self.staffnumber = staffid

    def getEmployeeName(self):
        return self.getFullName() + ", " +  self.staffnumber
```



```
>>> x = Person("Bart", "Simpson")
>>> y = Employee("Homer", "Simpson", "231")
>>> x.getFullName()
'Bart Simpson'
>>> y.getFullName()
'Homer Simpson'
>>> y.getEmployeeName()
'Homer Simpson, 231'
```

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

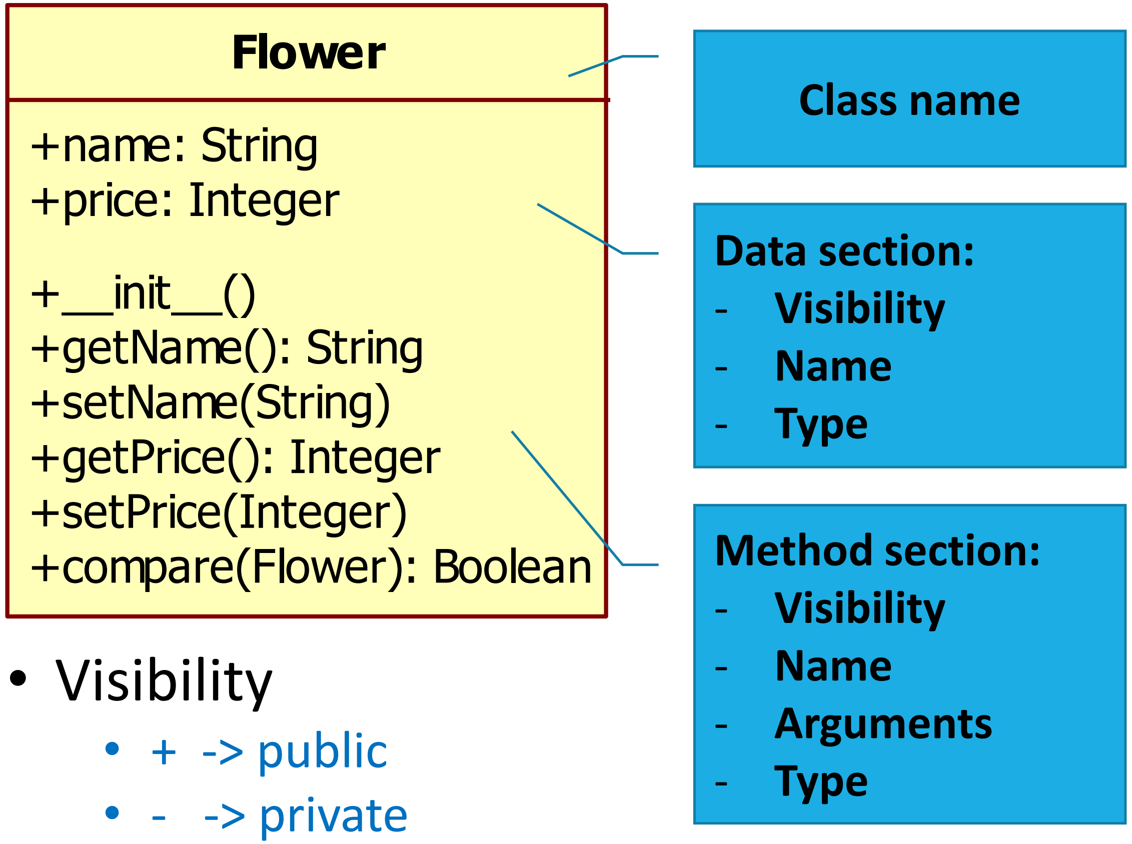
Unified Modelling Language (UML)

- 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

Class name

Data section:
- **Visibility**
- **Name**
- **Type**

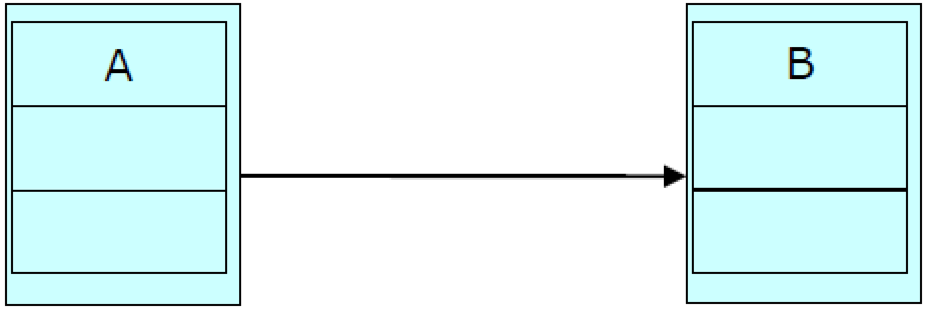
Method section:
- **Visibility**
- **Name**
- **Arguments**
- **Type**

```
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
```

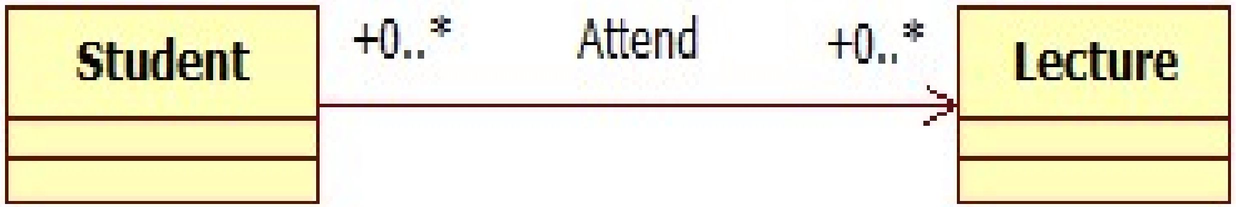
- Visibility
  - + -> public
  - - -> private
  - # -> protected

Relationships between classes: Association

- Association
  - Class A uses class B
  - Objects of A are connected to objects of B

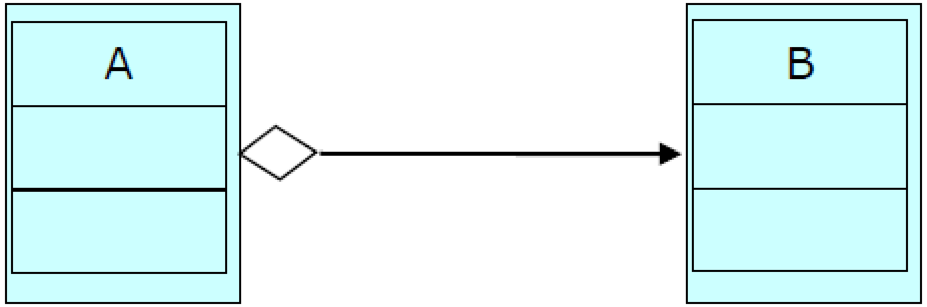


- An association can be named
- The ends of an association can be annotated with role names, ownership indicators, multiplicity, visibility
- Association can be bi-directional as well as uni-directional

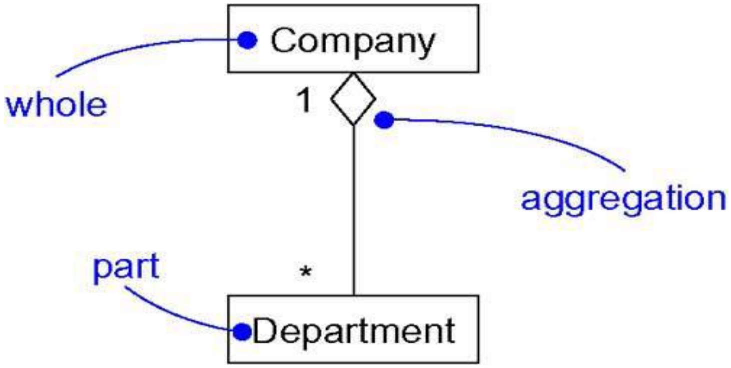


Relationships between classes: Aggregation

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

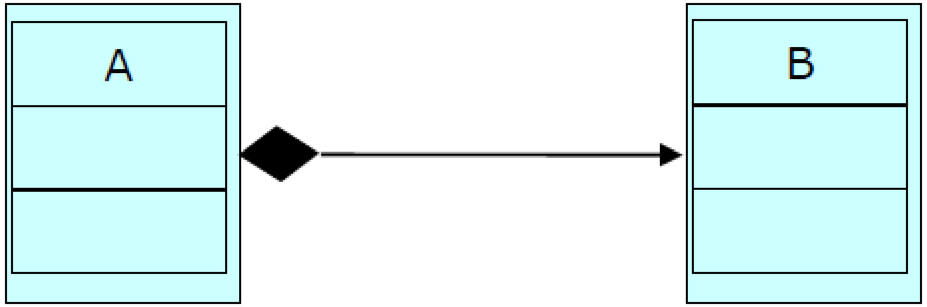


- Special kind of association used to model a „whole/part” relationship

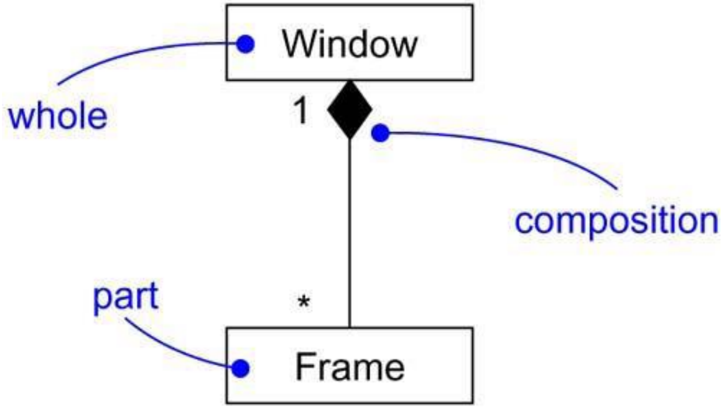


Relationships between classes: Composition

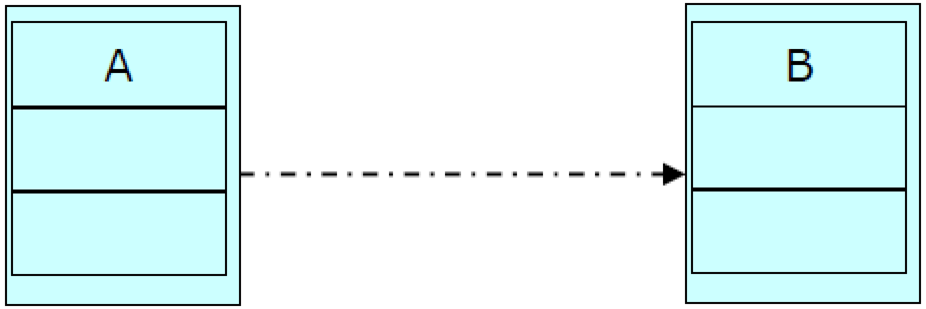
- Composition
  - A contains 1 or more B
  - B is created by A



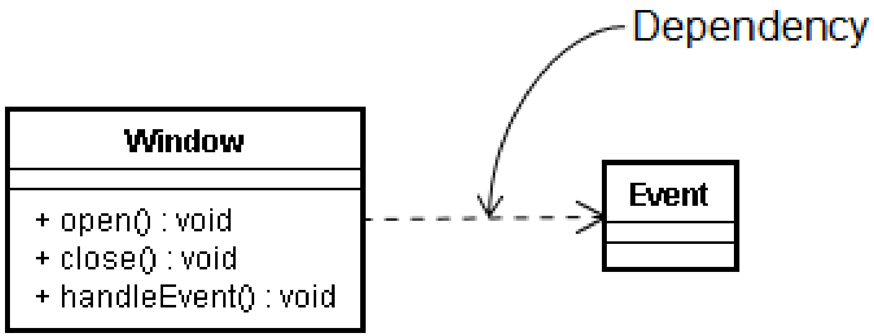
- Variation of simple aggregation: introduces a strong ownership and coincident lifetime as part of the whole



Relationships between classes: Dependency

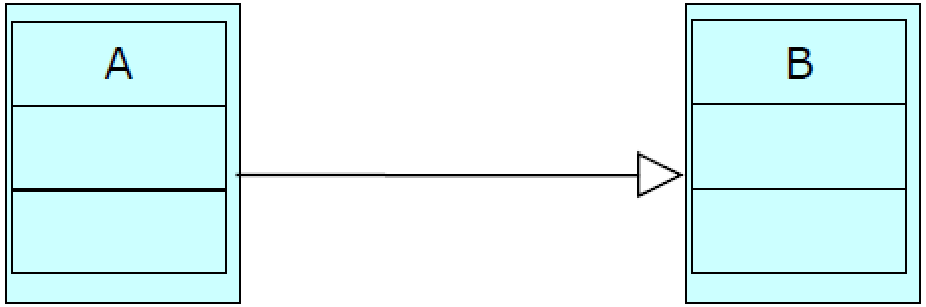


- Dependency
  - A depends on B
- Shows that:
  - one class uses operations from another class, or
  - it uses variables or arguments typed by the other class
  - if the used class changes => the operation of the other class may be affected

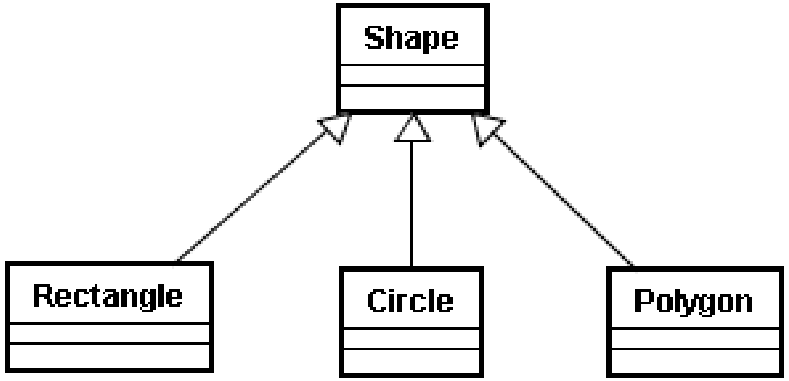


Relationships between classes: Generalization

- 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

NumPy

- Acronym for “Numerical Python”
- Useful to perform mathematical and logical operations on arrays
- Library
  - Multidimensional arrays and matrices
  - Collection of routines for processing arrays
- SciPy (Scientific Python) extends NumPy

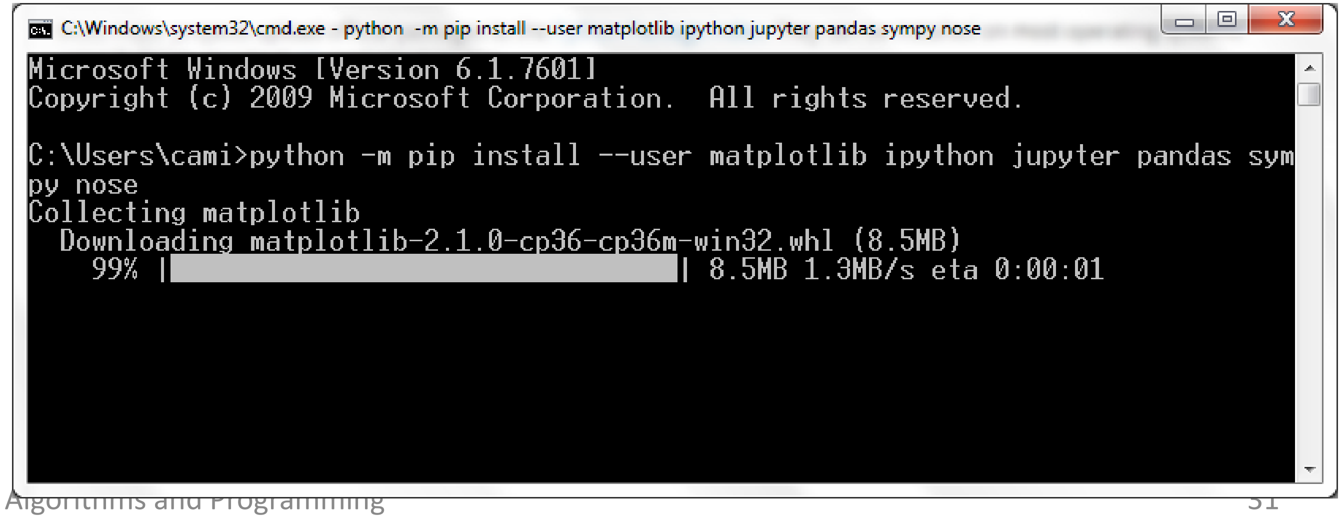
NumPy

- Has to be installed

  http://www.numpy.org

- Free Python distributions with SciPy for Windows
  - Anaconda, Canopy, Python(x,y)
- Installing via pip (Python’s standard package manager)
  - Need to have Python and pip already installed

```
python install numpy
```

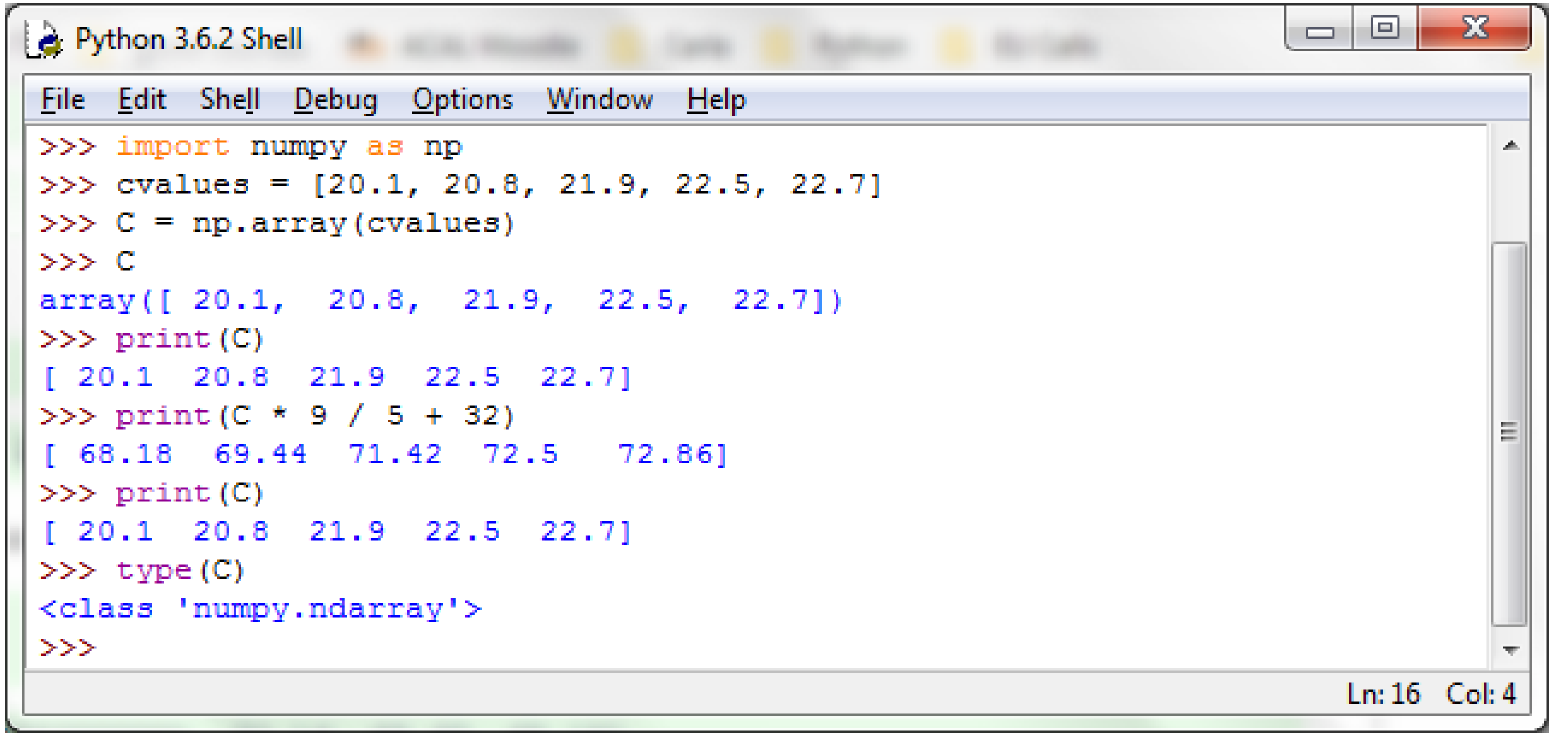


C:\Windows\system32\cmd.exe - python -m pip install --user matplotlib ipython jupyter pandas sympy nose

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\cami>python -m pip install --user matplotlib ipython jupyter pandas sym
py nose
Collecting matplotlib
  Downloading matplotlib-2.1.0-cp36-cp36m-win32.whl (8.5MB)
    99% |████████████████████████████████| 8.5MB 1.3MB/s eta 0:00:01
```

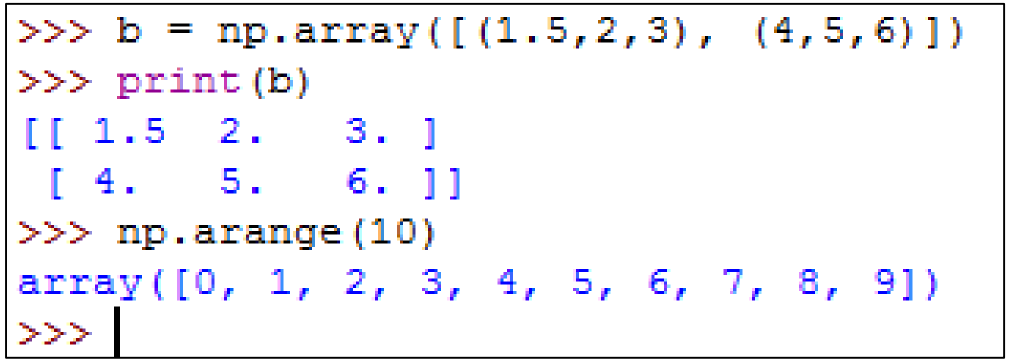
NumPy

```
import numpy
import numpy as np
```

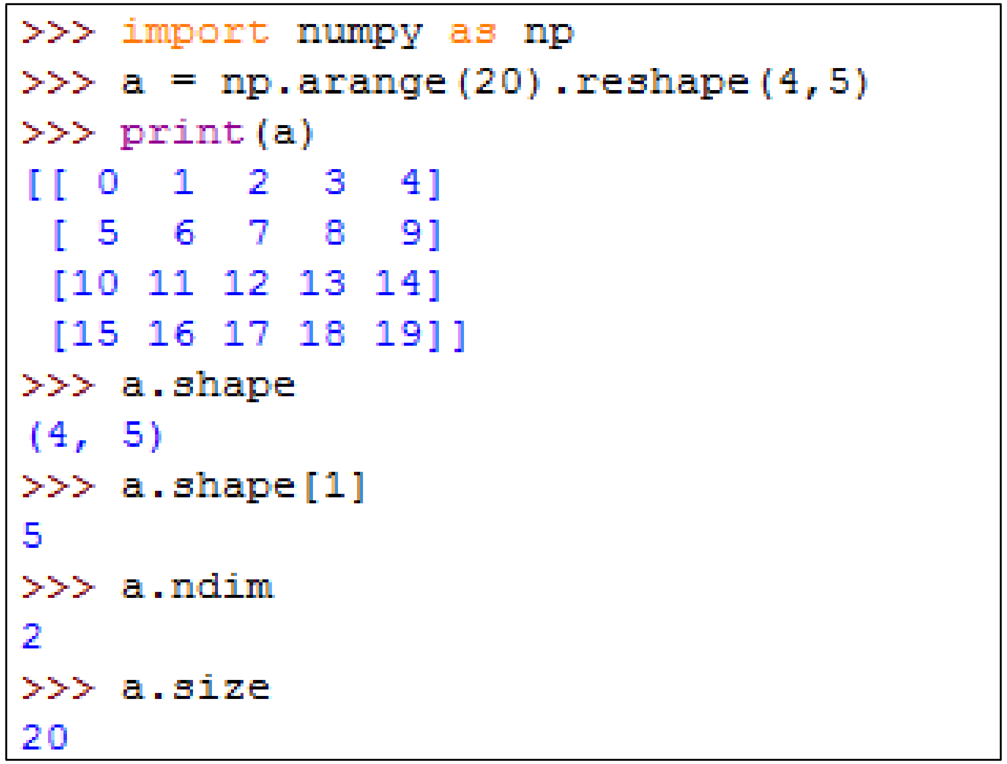


```
>>> import numpy as np
>>> cvalues = [20.1, 20.8, 21.9, 22.5, 22.7]
>>> C = np.array(cvalues)
>>> C
array([ 20.1,  20.8,  21.9,  22.5,  22.7])
>>> print(C)
[ 20.1  20.8  21.9  22.5  22.7]
>>> print(C * 9 / 5 + 32)
[ 68.18  69.44  71.42  72.5   72.86]
>>> print(C)
[ 20.1  20.8  21.9  22.5  22.7]
>>> type(C)
<class 'numpy.ndarray'>
>>>
```

NumPy

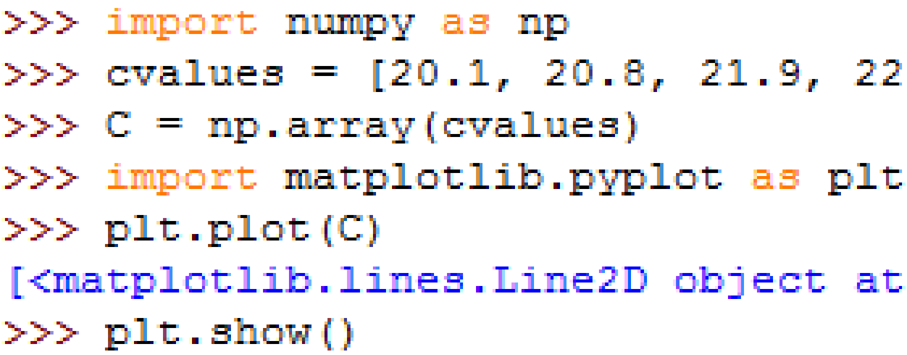


```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> print(b)
[[ 1.5  2.   3. ]
 [ 4.   5.   6. ]]
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> 
```

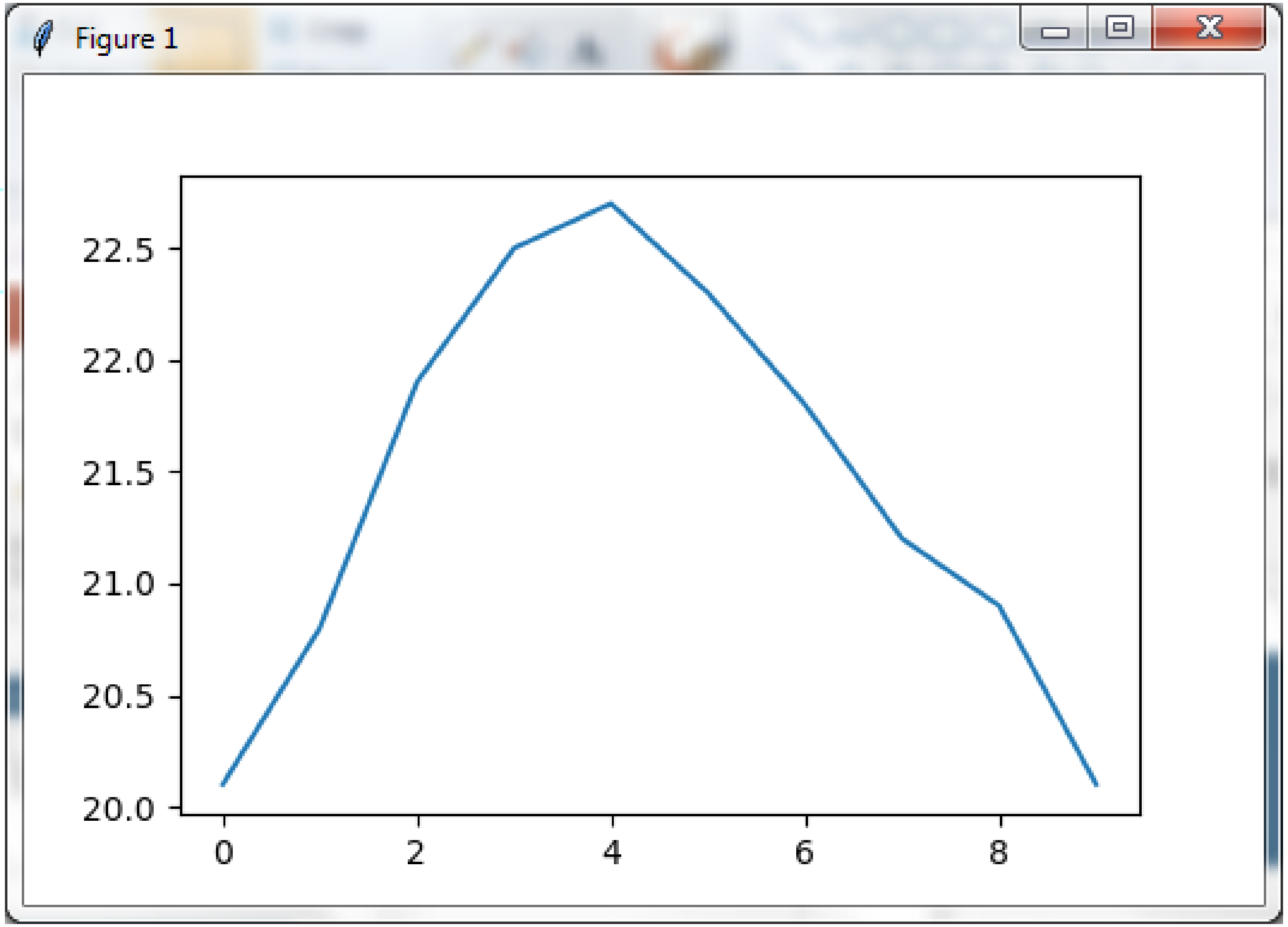


```
>>> import numpy as np
>>> a = np.arange(20).reshape(4,5)
>>> print(a)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
>>> a.shape
(4, 5)
>>> a.shape[1]
5
>>> a.ndim
2
>>> a.size
20
```

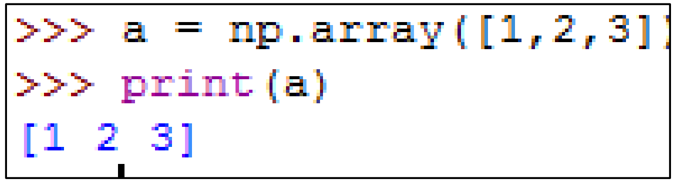
NumPy



```
>>> import numpy as np
>>> cvalues = [20.1, 20.8, 21.9, 22
>>> C = np.array(cvalues)
>>> import matplotlib.pyplot as plt
>>> plt.plot(C)
[<matplotlib.lines.Line2D object at
>>> plt.show()
```

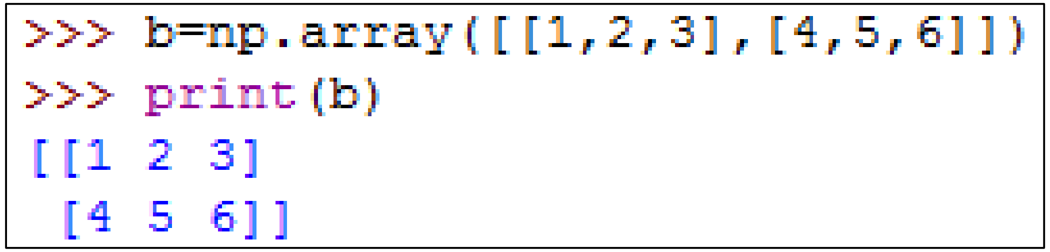


NumPy: Array creation

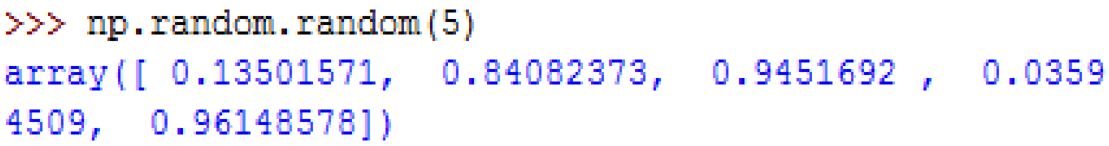


```
>>> a = np.array([1,2,3])
>>> print(a)
[1 2 3]
```

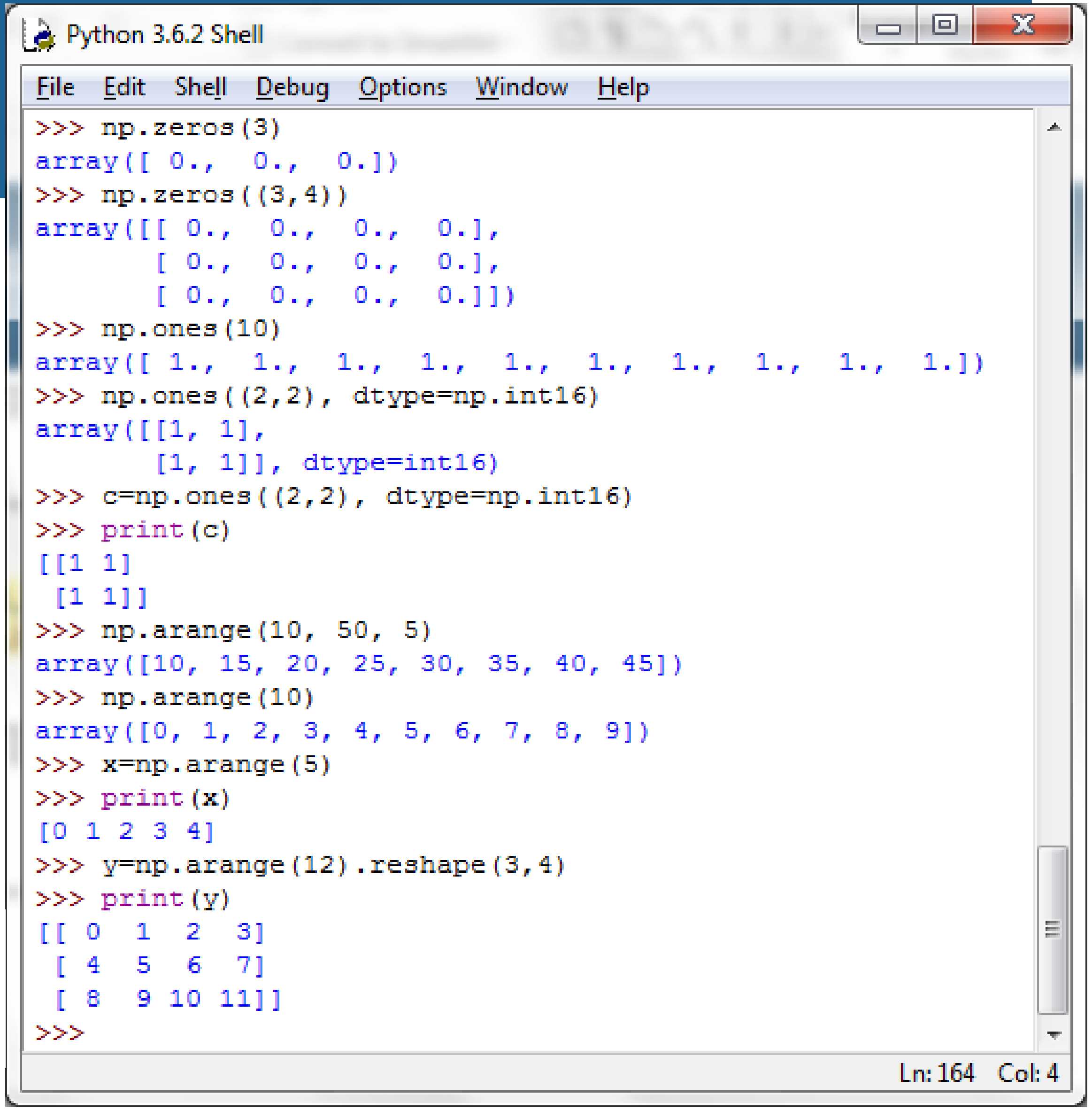
```
a = np.array(1,2,3,4)      # WRONG
a = np.array([1,2,3,4])    # RIGHT
```



```
>>> b=np.array([[1,2,3],[4,5,6]])
>>> print(b)
[[1 2 3]
 [4 5 6]]
```

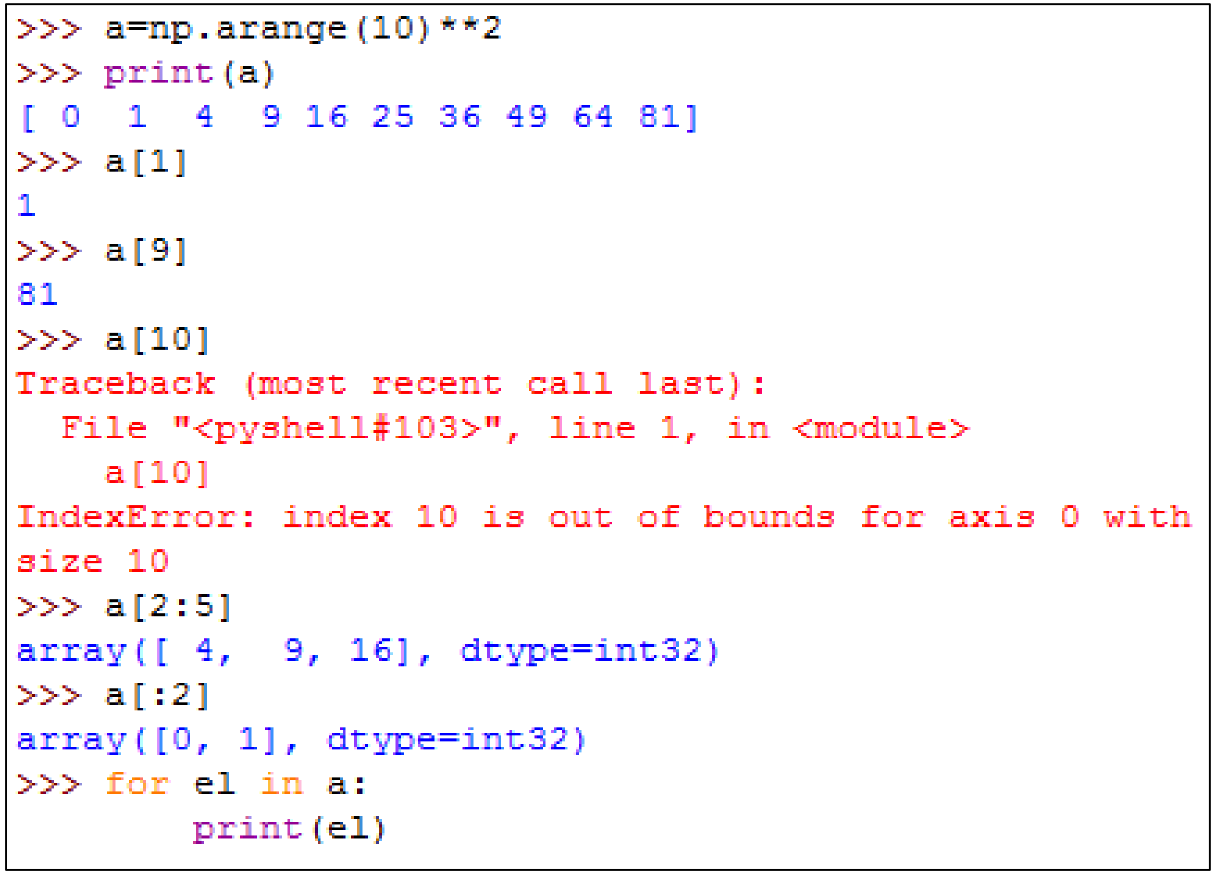


```
>>> np.random.random(5)
array([ 0.13501571,  0.84082373,  0.9451692 ,  0.0359
4509,  0.96148578])
```

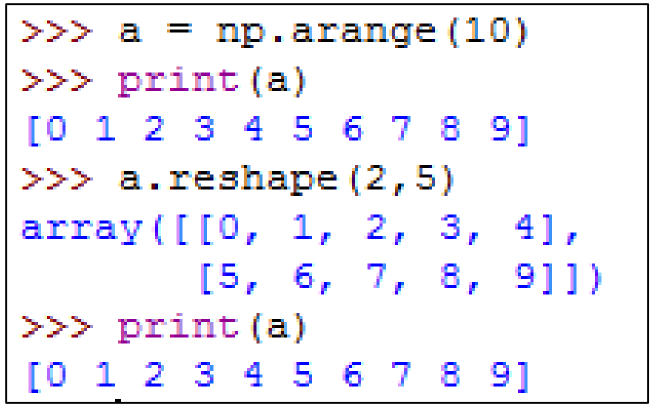


```
>>> np.zeros(3)
array([ 0.,  0.,  0.])
>>> np.zeros((3,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones(10)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> np.ones((2,2), dtype=np.int16)
array([[1, 1],
       [1, 1]], dtype=int16)
>>> c=np.ones((2,2), dtype=np.int16)
>>> print(c)
[[1 1]
 [1 1]]
>>> np.arange(10, 50, 5)
array([10, 15, 20, 25, 30, 35, 40, 45])
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x=np.arange(5)
>>> print(x)
[0 1 2 3 4]
>>> y=np.arange(12).reshape(3,4)
>>> print(y)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>>
```

NumPy: Basic operations

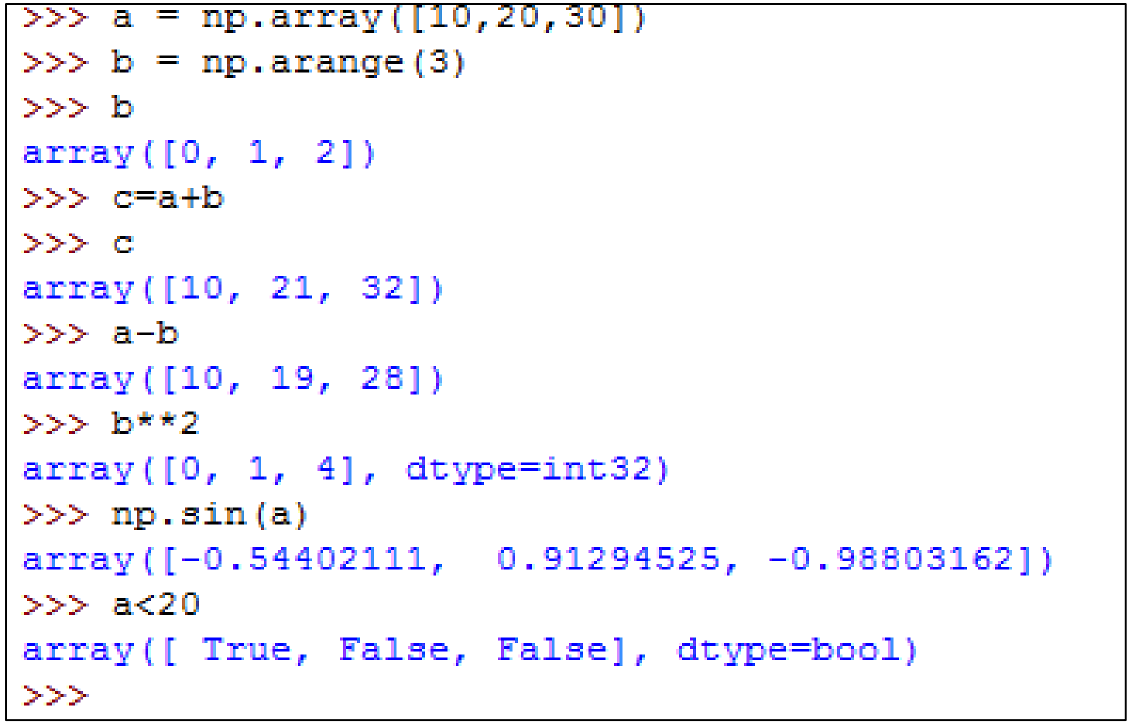


```
>>> a=np.arange(10)**2
>>> print(a)
[ 0  1  4  9 16 25 36 49 64 81]
>>> a[1]
1
>>> a[9]
81
>>> a[10]
Traceback (most recent call last):
  File "<pyshell#103>", line 1, in <module>
    a[10]
IndexError: index 10 is out of bounds for axis 0 with
size 10
>>> a[2:5]
array([ 4,  9, 16], dtype=int32)
>>> a[:2]
array([0, 1], dtype=int32)
>>> for el in a:
        print(el)
```



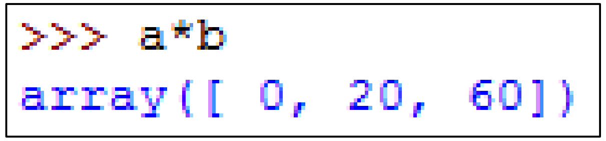
```
>>> a = np.arange(10)
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]
>>> a.reshape(2,5)
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]
```

NumPy: Basic operations



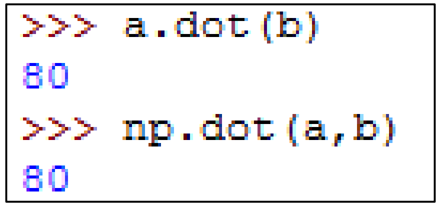
```
>>> a = np.array([10,20,30])
>>> b = np.arange(3)
>>> b
array([0, 1, 2])
>>> c=a+b
>>> c
array([10, 21, 32])
>>> a-b
array([10, 19, 28])
>>> b**2
array([0, 1, 4], dtype=int32)
>>> np.sin(a)
array([-0.54402111,  0.91294525, -0.98803162])
>>> a<20
array([ True, False, False], dtype=bool)
>>>
```

Elementwise product



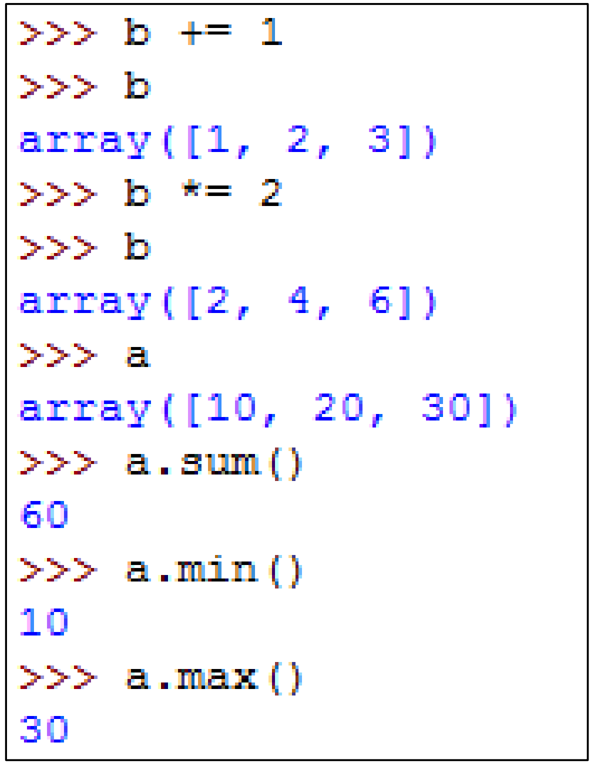
```
>>> a*b
array([ 0, 20, 60])
```

Array product

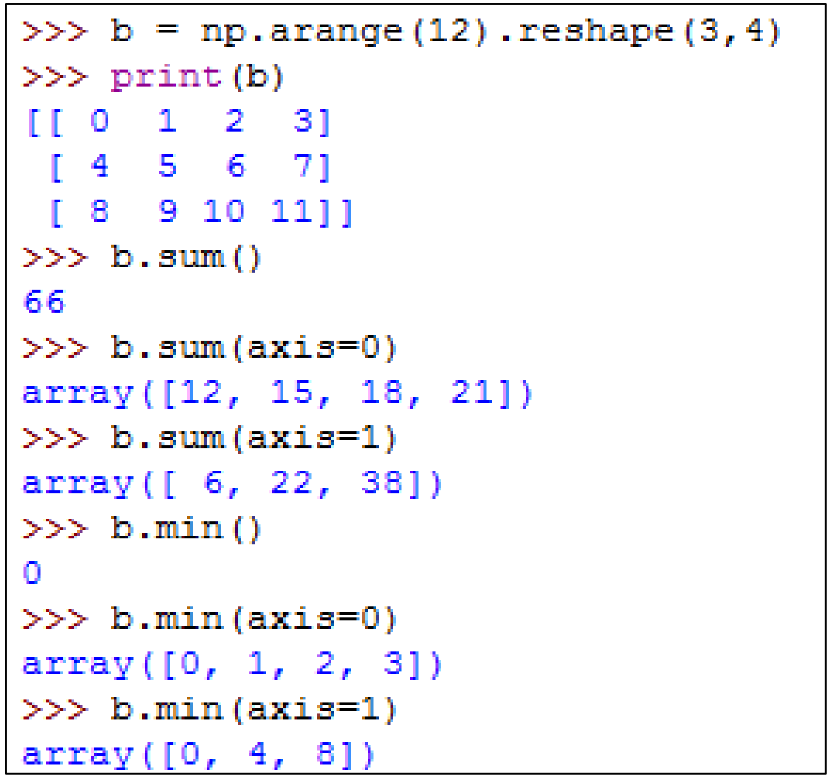


```
>>> a.dot(b)
80
>>> np.dot(a,b)
80
```

NumPy: Basic operations



```
>>> b += 1
>>> b
array([1, 2, 3])
>>> b *= 2
>>> b
array([2, 4, 6])
>>> a
array([10, 20, 30])
>>> a.sum()
60
>>> a.min()
10
>>> a.max()
30
```



```
>>> b = np.arange(12).reshape(3,4)
>>> print(b)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> b.sum()
66
>>> b.sum(axis=0)
array([12, 15, 18, 21])
>>> b.sum(axis=1)
array([ 6, 22, 38])
>>> b.min()
0
>>> b.min(axis=0)
array([0, 1, 2, 3])
>>> b.min(axis=1)
array([0, 4, 8])
```

Next time

- Testing and debugging
- Recursivity

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.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. 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

Bibliography

The content of this course has been prepared using the reading materials from previous slide, different sources from the Internet as well as lectures on Fundamentals of Programming held in previous years by:

- Prof. Dr. Laura Dioşan - www.cs.ubbcluj.ro/~lauras
- Conf. Dr. Istvan Czibula - www.cs.ubbcluj.ro/~istvanc
- Lect. Dr. Andreea Vescan - www.cs.ubbcluj.ro/~avescan
- Lect. Dr. Arthur Molnar - www.cs.ubbcluj.ro/~arthur