

ComIT

00P

THE TOPICS

- Classes
- Members
 - ✓ Attributes
 - ✓ Methods
 - ✓ Messages
 - ✓ Attributes and Local Variables
 - ✓ Member Access Operator
 - ✓ Members Accessibility
 - ✓ Encapsulation
 - ✓ Interface

INTRODUCTION (I)

OOP uses the same basics as we've been using

- We change the way of thinking for solutions of our problems.
 - ✓ Instead of emphasizing on separating the data from the operations that manipulate these data
 - ✓ We proceed to consider the data and operations that manipulate them as encapsulated in objects.

INTRODUCTION (II)

- OOP is structured and modular
 - ✓ There is a disciplined style of writing in programming
 - ✓ Separate different parts of the program into modules (Classes).

INTRODUCTION (III)

- From now on we are going to dedicate ourselves to studying the aspects and the tools that the OOP offers us to face the task of realizing programs
 - ✓ We are leaving the “main script”
 - ✓ Let's think about classes and objects, distributing algorithms and data along the methods and attributes.

CLASSES (I)

- Theoretical Definition: A class is a concept that encompasses a set of objects with similar characteristics.
- Practical Definition: A class is a "template" from which we create objects
 - ✓ In this template we define the common characteristics

WHY A TEMPLATE?

- Let's consider "Bicycle" class
 - ✓ I can create 1, 10, 2000 bikes
 - ✓ Each with its state.
 - ✓ If I had to describe 2000 bikes separately (without the template) I would have a lot of work to do.
- **We design once and construct as many objects as we want to.**

CLASSES (II)

- Design a class means defining:
 - ✓ **Attributes**
 - ✓ **Methods**
 - ✓ How to **construct objects**

CLASSES (III)

- A class in Python is saved in a text file (which we can open with a text editor and see its contents perfectly).
- In Python, one file is called a module. A module can consist of multiple classes or functions.
- One file (module) should contain classes / functions that belong together, i.e. provide similar functionality or depend on each other.
- Of course, we should not exaggerate this. Readability really suffers if our module consist of too many classes or functions. Then it is probably time to regroup the functionality into different modules and create packages.

CLASS STRUCTURE

Example:

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
```

CLASS STRUCTURE

- Example:
`class Parrot:`
- The word Parrot is an identifier that we define to be able to refer to the class that surrounds the different objects parrots.
- The body of the class is tabulated to the right

CLASS STRUCTURE

- Body of the class is like this:
 - ✓ Members
 - ✓ Attributes
 - ✓ Methods
 - ✓ Constructors

ATTRIBUTES (I)

The properties or characteristics of an object are stored in variables that are known as **attributes**.

ATTRIBUTES (II)

- Class variable – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- Instance variable – A variable that is defined inside a method and belongs only to the current instance of a class.
- The value of all the attributes of an object in an instant determines its state.

ATTRIBUTES (III)

Syntax:

```
class XX
```

```
    <IDENTIFIER VAR> = <EXPR>
```

ATTRIBUTES (IV)

Example:

```
class Parrot:
```

```
    # class attribute  
    species = "bird"
```

```
    # instance attribute  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

- Thus each particular parrot object will have its species, its name and its age.
- Each instance has its own set of attributes with its particular values.
- Each one has its own state.

METHODS (I)

- In Theory
 - ✓ Methods are used to define the behavior of the objects of a class.
- In Reality
 - ✓ The "places" where sentences are written to handle instructions and data
 - ✓ The **algorithms**

METHODS (II)

Syntax:

```
class Parrot:
    # class attribute
    species = "bird"

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)
```

- Similar to pseudo code subalgorithms.
 - ✓ Wrap statements
 - ✓ Have parameters
 - ✓ Have a return value (optional)

METHODS (III)

Example:

```
class Addition:
    first = 0
    second = 0
    answer = 0

    # parameterized constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two numbers = " + str(self.answer))

    def calculate(self):
        self.answer = self.first + self.second
```

METHODS (III - CONT.)

Example:

```
# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)

# perform Addition
obj.calculate()

# display result
obj.display()
```

MESSAGES (I)

- Methods are executable portions of code
- When a method of an object is invoked it is said that a message or request is sent to that object to perform that action.

MESSAGES (II)

- A message is composed of:
 - ✓ Receiving Object
 - ✓ Method that is executed
 - ✓ Arguments (if required)

- Examples:

`obj.calculate() # message perform Addition`

`obj.display() # message display result`

METHODS + ATTRIBUTES (I)

```
class Addition:
    first = 0
    second = 0
    answer = 0
    def calculate(self):
        self.answer = self.first + self.second
```

“answer”, “first” and “second” seem like undeclared variables in calculate.

But it works! How?

METHODS + ATTRIBUTES (II)

```
class Addition:  
    first = 0  
    second = 0  
    answer = 0  
    def calculate(self):  
        self.answer = self.first + self.second
```

By “answer”, “first” and “second” I’m talking about the attribute of the object I’m “calculating”.

METHODS + ATTRIBUTES (III)

- Method = **Object** Functionality
 - ✓ Invoking a method = ask an **object** to execute a functionality.
- To invoke a method requires a concrete object, which is exactly the one that we asked to perform this function.

METHODS + ATTRIBUTES (IV)

- Theory
 - ✓ Within the method there is the object that executes the statements and this has knowledge of its attributes.
- Reality
 - ✓ Within a method that has an object you can access any of its attributes.
 - ✓ We can see the methods as operations that modify data that are stored in the attributes.

METHODS WITH RETURN VALUE

- If the method returns a value, then the last line in it has to use a special statement that indicates that value is returned.

return <EXPRESSION>

METHODS WITH RETURN VALUE

```
# instance method
def sing(self, song):
    return "{} sings {}".format(self.name, song)

def dance(self):
    return "{} is now dancing".format(self.name)
```

FIRST CLASS

```
#!/usr/bin/python
```

```
class Employee:
```

```
    #Common base class for all employees
```

```
    empCount = 0
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
        Employee.empCount += 1
```

```
    def displayCount(self):
```

```
        print "Total Employees %d" % Employee.empCount
```

```
    def displayEmployee(self):
```

```
        print "Name : ", self.name, ", Salary: ",  
self.salary
```

AND NOW WE USE IT

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

A CLASS AS A TEMPLATE OF OBJECTS

- Each employee has its own name and salary!
- Remember: each object has its **state**

PARAMETERS AND ARGUMENTS

- Parameters are variables of a method that will assume a value when the method is invoked.
- The value that is actually assigned to a parameter of a method when invoked is called argument.
 - ✓ This value is the result of an expression that returns a value of the same type as the parameter.
- The order in which the parameters are declared and the arguments are passed establishes the correspondence between one and the other.

PARAMETERS AND ARGUMENTS

- The idea of parameterizing is to extend or generalize the utility of a method, to suit different uses.

PARAMETERS AND ARGUMENTS

Example:

```
def increaseSalary(self):  
    self.salary+=10
```

vs.

```
def increaseSalary(self,howMuch):  
    self.salary+=howMuch
```

- With the second one we can
 - ✓ Specify how much we want to increase the salary
 - ✓ We can even decrease it (if howMuch < 0)

PARAMETERS AND ARGUMENTS

Parameters are local variables within the method (or constructor) in which they are defined. They are used like any other type of variable, knowing that once the method is finished, they die (unless they are of reference).

CONSTRUCTOR (I)

- Before you can use an object it is mandatory to build it.
- To build objects we use constructors.
- In practical terms: a constructor is a special method that serves to initialize the state of an object.

CONSTRUCTOR (II)

- It is declared and defined as any other method only that the name is `__init__`

```
class Employee:
    #Common base class for all employees
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
```

CONSTRUCTOR (II)

- If a constructor is not provided, Python provides a default constructor with no parameters.
- If a constructor is defined (with or without parameters) then Python does nothing for us and we must use only the defined one (s).
- Unlike Java, we cannot define multiple constructors. However, you can define a default value for a parameter if one is not passed.

USING THE CONSTRUCTOR

- How do I invoke the constructor to create objects?
- It is invoked when we create a **new** object using the constructor name.

```
emp1 = Employee("Zara", 2000)
```

‘ SELF ’

- Within an instance method or a constructor, “self” is a variable that saves the reference to the current object
 - ✓ The one whose method is being invoked
 - ✓ The one that is being built
- The most common use of self is because the name of an attribute is hidden by a parameter of the same name.

‘ SELF ’

Example:

```
def __init__(self, name, salary):  
    self.name = name  
    self.salary = salary  
    Employee.empCount += 1
```

how do I differentiate attribute “name”
from parameter “name”?

SOME PRACTICE

- Add some new attributes to our Employee:
 - ✓ Address (in any format you want)
 - ✓ SIN #
 - ✓ Job/Role
 - ✓ Date of Birth
- Add some new methods and use them
 - ✓ Increase/Change Salary
 - ✓ Show Employee information (choose the format)
 - ✓ Edit Employee information

SHOWING OBJECT STATE

- A recurring issue will be to report by console the state of an object
- The print (<Var>) prints the contents of a variable
 - ✓ Primitive data type
 - ✓ Value that represents the data
 - ✓ Class type
 - ✓ Value that represents the ID of the object
 - ✓ Not the state
- So?

SOLUTION 1

- We can use methods to look for individual values of attributes:

```
def displayEmployeeName(self):  
    print "Name : ", self.name
```

```
def displayEmployeeName(self):  
    print "Salary: ", self.salary
```

- Redundant code

SOLUTION 2

- Create a method that returns the state

```
def displayEmployee(self):  
    print "Name : ", self.name, ", Salary: ", self.salary
```

- Other names:
 - ✓ getState
 - ✓ toString
- Note: you can also try **str(obj)** method

ATTRIBUTES AND LOCAL VARIABLES

- An attribute is a declared variable within the body of a method of a class
- A local variable is a declared variable within the body of a method or a parameter of the same.

ATTRIBUTES AND LOCAL VARIABLES

```
class Employee:
    #Common base class for all employees
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def increaseSalary(self, howMuch):
        salary = 0
        salary += howMuch # assigning 0 to local
        variable and not attribute!
        print(self.salary)
        print(salary)
```

Rule: Local variables hide attributes!

EXERCISE

- Let's build a Car class, for every car in the world.
 - ✓ Attributes?
 - ✓ ¿Methods?
- Let's use it in.

USING OBJECTS

- We have already seen that in the variables of a class type we keep references to objects.
- Now how do I use these objects??
 - ✓ How do I send messages?
 - ✓ How do I access its attributes?
- How do I access the object's members?.

MEMBER ACCESS OPERATOR

- The dot “.” is the access operator for the object’s members.
- We need 3 things to use objects:
 - ✓ **Reference**
 - ✓ **Operator** “.”
 - ✓ **Name** of the member
 - ✓ Attribute
 - ✓ Or method and its arguments

QUICK QUESTION

Can we access all the members?

ANSWER

No!

This depends from where you want to access and the visibility that the member has.

ACCESSING MEMBERS

The access modifiers in Python are used to modify the default scope of variables. There are three types of access modifiers in Python: public, private, and protected.

- ✓ **Private(__)**: private variables can only be accessed inside the class.
 - ✓ **public**: variables with public access modifiers can be accessed anywhere inside or outside the class.
 - ✓ **Protected(_)**: protected variables can be accessed within the same package
- Visibility is a member modifier.

APPLYING MODIFIERS

```
class Car:
    def __init__(self):
        print ("Engine started")
        self.name = "corolla"
        self.__make = "toyota"
        self._model = 1999
```

- Private attribute “make”
- Public attribute “name”
- Protected attribute “model”

UNDERSTANDING MODIFIERS

```
car_a = Car()
print(car_a.name)
print(car_a.make)
```

- Can not access attribute make outside the class because it is private
- The attribute name can be accessed because it is public.
- **Note:** the same modifier can be applied to methods!!!!

ACCESSING MEMBERS

```
b1 = Car()  
b1.make = 10  
b1.displayMake()
```

```
class Car:  
    def __init__(self):  
        print ("Engine started")  
        self.name = "corolla"  
        self.__make = "toyota"  
        self._model = 1999  
  
    def displayMake(self):  
        print(self.make)
```

- Private Attributes
- Public Methods

ACCESSING MEMBERS

```
b1 = Car()  
b1.make = 10  
b1.displayMake()
```

```
class Car:  
    def __init__(self):  
        print ("Engine started")  
        self.name = "corolla"  
        self.make = "toyota"  
        self._model = 1999  
  
    def __displayMake(self)  
        print(self.make)
```

- Public Attributes
- Private Methods

ACCESSING MEMBERS

```
b1 = Car()  
b1.make = 10  
b1.displayMake()
```

```
class Car:  
    def __init__(self):  
        print ("Engine started")  
        self.name = "corolla"  
        self.__make = "toyota"  
        self._model = 1999  
  
    def __displayMake(self)  
        print(self.make)
```

- Private Attributes
- Private Methods

ACCESSING MEMBERS

```
b1 = Car()  
b1.make = 10  
b1.displayMake()
```

```
class Car:  
    def __init__(self):  
        print ("Engine started")  
        self.name = "corolla"  
        self.make = "toyota"  
        self._model = 1999  
  
    def displayMake(self)  
        print(self.make)
```

- Public Attributes
- Public Methods

ACCESSING MEMBERS

- And why do I complicate it with access permissions?
- Why do I not make them all public?

ACCESSING MEMBERS

- What happens if we we assign to the public attribute “make” of the car the value “Honda”. Now we have a Honda Corolla.
- While the code is correct and the program would compile, and even execute, but ...
- I'm setting an invalid state.

ACCESSING MEMBERS

- If I deny the access to its attributes, and I do it indirectly through specific methods, I can control people not to assign invalid values.

ENCAPSULATION

- These access-to-members permissions are the foundation of the encapsulation mechanism, which is to hide details of the implementation
 - ✓ We use methods to manage the state of an object and do not modify it directly using its attributes.
- It has the following benefits...

ENCAPSULATION: BENEFITS

- We do not have to know the internal rules or the logic that an object carries, we only use it and we know that the methods will do the work we need.
- Example:
 - ✓ **print();**
 - ✓ Does any of you know how it works internally?

ENCAPSULATION: BENEFITS

- Avoid duplicating and spreading the code that corresponds to the same rule.
- You can use an internal method that calculates a specific logic and then reuse it in many other methods/algorithms

ENCAPSULATION: BENEFITS

- Everything goes in one place : **the method**
- “one ring to rule them all” – all the control logic for that specific task, goes where it has to go

ENCAPSULATION: BENEFITS

- If tomorrow the rules change, the only part that changes is the method.
 - ✓ Remaining unchanged the rest of the Program
 - ✓ Without this approach you would have to change every part of the code where you use it.

INTERFACE

- This is the part of the object that is visible to the rest of the objects.
 - ✓ That is, it is the set of visible members of an object that allow us to operate it.
- **Encapsulating involves exposing an interface.**

THE ART OF ENCAPSULATION

- Objects:
 - ✓ Has its own logic and rules.
 - ✓ In the Car class I will not include a method to calculate taxes.
 - ✓ We should not manipulate its details from the outside
 - ✓ Instead of:
 - ✓ `account.setBalance(account.getBalance() + amount) .`
 - ✓ We do:
 - ✓ `account.deposit(amount)`

THE ART OF ENCAPSULATION

- Designing a class is not trivial.
 - ✓ So let's have patience and practice.

EXERCISING

EXERCISING

THE END

QUESTIONS?

ComIT

00P II

THE TOPICS

- Grouping Objects
- List
 - ✓ Primitive types
 - ✓ Objects
- Grouping objects
 - ✓ Aggregation
 - ✓ Composition

GROUPING OBJECTS (I)

- So far we have been working on designing classes whose objects have attributes of a primitive data type.
- Can the type of the attribute be a class type?
 - ✓ Of course.

GROUPING OBJECTS (II)

- An object groups or gathers other objects.
 - ✓ A segment has two points
 - ✓ A faculty has a director and professors
 - ✓ A person has a nationality and speaks a language
 - ✓ An employee has life insurance.
 - ✓ A window has an area
 - ✓ And many funny other groupings....
- We manage to represent more complex realities, more accurately.

CODING RELATIONSHIPS

```
class Segment
    def __init__(self, origin, end):
        self._origin=origin #Point origin is an object
        self._end=end

class School
    def __init__(self, theBoss, theOneWhoWorks):
        self._theBoss=theBoss #theBoss is an object
        person
        self._theOnewhoWorks=theOnewhoWorks

# Other methods
```

WHY AN OBJECT AS ATTRIBUTE?

- To represent complex realities
 - ✓ More accurately and naturally.
 - ✓ Reducing the intrinsic difficulty involved.
- Still not convinced?

WITHOUT OBJECT GROUPING

```
class Segment
    def __init__(self, pxo,pyo,pzo,pxf,pyf,pzf) :
        self._pxo=pxo
        self._pyo=pyo
        self._pzo=pzo
        self._pxf=pxf
        self._pyf=pyf
        self._pzf=pzf

class School
    def __init__(self, nameDean,
ageDean,phoneDean,nameVD,ageVD,phoneVD) :
        self._nameDean=nameDean
        self._ageDean=ageDean
        self._phoneDean=phoneDean
        self._nameVD=nameVD
        self._ageVD=ageVD
        self._phoneVD=phoneVD
        # Other methods
```

WITHOUT OBJECT GROUPING

- The design of the class...
 - ✓ Gets bigger and complex
 - ✓ Less flexible
 - ✓ Attributes are repeated

REMEMBER

- An instance of the class Segment is linked with two instances of the class points.
- An instance of the school class with an instance of Dean ("John Doe") and another instance of ViceDean ("Jane Doe").
- An instance of Window ("The one I have on the right") with an area instance ("corresponds to the particular area that occupies it").
- ... and more...
- How do I link these instances?

LINKING OBJECTS

- Ways to link objects
 - ✓ **In the constructor**
 - and/or*
 - ✓ Using **Setters**

LINKING AT THE CONSTRUCTOR

```
class Segment
    def __init__(self, origin, end):
        self._origin=origin #Point origin is an object
        self._end=end
```

The way we already showed!

```
p1 = Point(0,0,0)
p2 = Point(5,5,0)
seg1 = Segment(p1,p2)
```

Eazy Peazy

DEFINING SETTERS

```
class Segment
    def __init__(self):
        self._origin=None
        self._end=None

    def setOrigin(self,origin)
        self._origin=origin

    def setEnd(self,end)
        self._end=end
```

We'll see other ways to use setter and getter in Python, but this is just a start!

LINKING WITH SETTERS

```
p1 = new Point (0,0,0)
p2 = new Point (5,5,0)
seg1 = Segment()
seg1.setOrigin(p1)
seg1.setEnd(p2)
```

LINKING OBJECTS (I)

- What's the best way?
 - ✓ For starters, it is good practice to define a constructor that is consistent with reality.
 - ✓ If an object can not exist without being related to others, then the constructor should articulate that
 - ✓ Setters are incorporated if you want to give the object the ability to change its links.

LISTS (I)

- Let's review our List definition.
- Finite set of elements storing data
 - ✓ It can be accessed by specifying an index.

0	1	2	3	4	5	...	N
						...	

LISTS (II)

- There is a difference between a data in the List whose base type is a primitive type than those that are an object type.
- We have to remember the difference that exists when a variable is of a primitive type or an object type.

PRIMITIVE TYPE LISTS

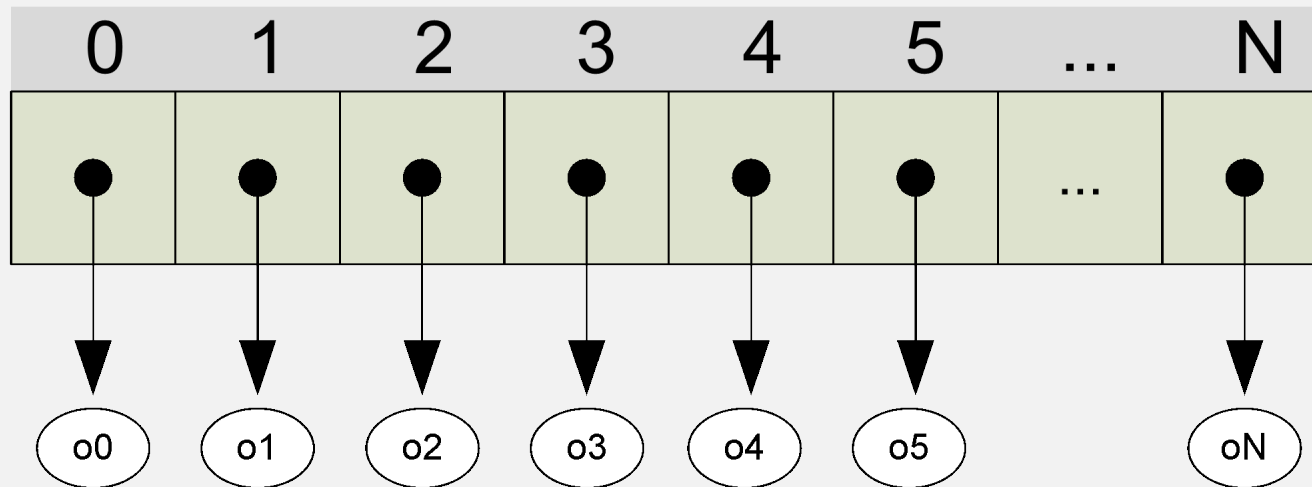
- Each element is a primitive type value

0	1	2	3	4	5	...	N
D0	D1	D2	D3	D4	D5	...	DN

- List of values

OBJECT TYPE LISTS

- Each element is a variable of class type.
 - ✓ **Reference to an object**



- **Objects' lists**

EXAMPLE WITH BICYCLE TYPE

- It's one thing to create an list with X bikes
 - ✓ **“The closet”** for the bikes
- Another thing is to create X bike objects
 - ✓ **X specific bikes**

CONCLUSION

- **Building the structure that stores the data is one thing.**
- **Populate said structure is something else.**

EXERCISE

- **Create a script.**
 - ✓ Create a Bicycle class
 - ✓ Let's build a list for 6 bikes
 - ✓ Accelerate the bicycles in the odd elements (1,3,5).
 - ✓ Let's print out the status of each bicycle referenced in that list.

GROUPING WITH LISTS

- Can we use a list as an attribute?
 - ✓ Of course!

GROUPING WITH ARRAYS

```
class Polygon
    def __init__(self, p1,p2,p3):
        self._points=[p1,p2,3]

    def addPoint(self,point):
        points.append(point)

    /* other members*/
```

WHY AN OBJECT LIST AS ATTRIBUTE?

- We can represent more complex realities when there is some object (Container) that gathers or groups a set of objects.
- Still not convinced?

WITHOUT GROUPING WITH LISTS

```
class Polygon
    def __init__(self, p1,p2,p3,...,pn):
        self._p1=p1
        self._p2=p2
        self._p3=p3
        ...
        self._pn=pn
```

WITHOUT GROUPING WITH LISTS

- This class design is impossible to use.

GROUPING WITH LISTS

- We saw how to design classes to contemplate grouping with lists.
- But, again, this is half the movie
 - ✓ We need to know how to use the objects of this class
 - ✓ In particular, when I want to link or articulate concrete objects with each other.

REMEMBER

- An instance of the polygon class groups several point instances.
- An instance of the current account class consists of several instances of operation
- An instance of the class university has several department instances.
- A library instance has multiple instances of books
- ... and more...
- How do I link these instances?

LINKING OBJECTS

- How to link objects
 - ✓ **In the constructor**
 - ✓ Build “the closet”
 - ✓ **Use of setters**
 - ✓ To set values of each element
 - ✓ Establish references

OTHER WAYS OF LINKING

- The approaches we saw to link one object group to another are of the **aggregation** type.
 - ✓ Objects are constructed outside the methods of the class.
- There are other ways of composition, where the container object is responsible for creating and managing its objects.
 - ✓ Objects are constructed within the methods of the class (not outside)
 - ✓ I have to initialize the list on the constructor
 - ✓ **Use convenient methods**
 - ✓ To add, retrieve, remove, and modify values that each element contains

COMPOSITION

```
class Polygon
    def __init__(self):
        self._points=[]

    def addPoint(self,point):
        points.append(point)
    def getPoint (...)
    def updatePoint (...)
    def removePoint (...)

    /* other members*/
```

ARRAY COMPOSITION

- The trick is to create the empty list and update it.
- As soon as the list is constructed, we consider that we don't have any element.
 - ✓ 0 elements in use.

ADDING VALUES

- The new value is stored in the element #1.
- The length of our list is incremented by 1

REMOVING VALUES

- The length is decremented by 1.
- We can use “pop” or “remove”

MODIFYING VALUES

- The value in an element is modified
- You must verify that the item contains a consistent value with what we want to represent

COMPOSITION AND AGGREGATION

- When an object groups or gathers a set of other objects, there are two ways of relating them:
- **Aggregation**
 - ✓ No existential dependency
 - ✓ Objects can be shared
- **Composition**
 - ✓ Existential dependency
 - ✓ Exclusive Objects
 - ✓ If the father die, the children die

COMPOSITION AND AGGREGATION

- **Aggregation**

- ✓ Constructor + setter
- ✓ Objects created outside container object methods

- **Composition**

- ✓ Constructor + methods
- ✓ Objects created within container object methods

COMPOSITION AND AGGREGATION

Here is a complex example:

```
class Student:
```

```
    def __init__(self, name, student_number):
        self.name = name
        self.student_number = student_number
        self.classes = []
```

```
    def enrol(self, course_running):
        self.classes.append(course_running)
        course_running.add_student(self)
```

```
class Department:
```

```
    def __init__(self, name, department_code):
        self.name = name
        self.department_code = department_code
        self.courses = {}
```

```
    def add_course(self, description, course_code, credits):
        self.courses[course_code] = Course(description, course_code, credits, self)
        return self.courses[course_code]
```

COMPOSITION AND AGGREGATION

```
class Course:
    def __init__(self, description, course_code, credits, department):
        self.description = description
        self.course_code = course_code
        self.credits = credits
        self.department = department
        self.department.add_course(self)

        self.runnings = []

    def add_running(self, year):
        self.runnings.append(CourseRunning(self, year))
        return self.runnings[-1]

class CourseRunning:
    def __init__(self, course, year):
        self.course = course
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)
```

COMPOSITION AND AGGREGATION

- A student can be enrolled in several courses (CourseRunning objects), and a course (CourseRunning) can have multiple students enrolled in it in a particular year, so this is a many-to-many relationship. A student knows about all his or her courses, and a course has a record of all enrolled students, so this is a bidirectional relationship. These objects aren't very strongly coupled – a student can exist independently of a course, and a course can exist independently of a student.
- A department offers multiple courses (Course objects), but in our implementation a course can only have a single department – this is a one-to-many relationship. It is also bidirectional. Furthermore, these objects are more strongly coupled – you can say that a department owns a course. The course cannot exist without the department.
- A similar relationship exists between a course and its “runnings”: it is also bidirectional, one-to-many and strongly coupled – it wouldn't make sense for “MAM1000W run in 2013” to exist on its own in the absence of “MAM1000W”.

PRACTICE

PRACTICE

THE END

QUESTIONS?

ComIT

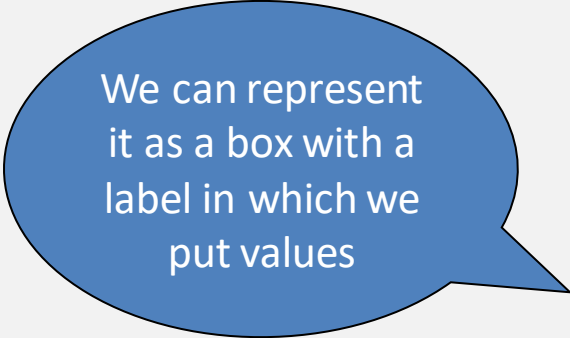
00P III

THE TOPICS

- Variable contents
 - ✓ Value and Reference Variables
 - ✓ Assignment =
 - ✓ Comparison ==
 - ✓ Memory management
- Sending parameters

VARIABLE'S CONTENT

- Let's review what a **variable** is
 - ✓ They have an identifying name
 - ✓ They have a content.
 - ✓ Contents are the stored value
 - ✓ It is important to know what the variables are storing, this depends on the type of data of the variable.
 - ✓ Remembering...



We can represent
it as a box with a
label in which we
put values

Content

CONTENT ACCORDING TO TYPE

- Variables of a primitive data type
 - ✓ **Their value directly represents the data they are storing.**
 - ✓ Value Variables
- Variables of a class type
 - ✓ **Its value is a reference to an object.**
 - ✓ Reference Variables

EXPRESSIONS BY TYPE

- In turn, the expressions can be classified into two types depending on what type of value they return when evaluating.
 - ✓ Primitive value
 - ✓ Variables, Constants, Literals (of a type of primitive data)
 - ✓ Invocations to methods with primitive data type returns
 - ✓ Combination of the above with operators
 - ✓ Reference
 - ✓ Variables, Constants, Literals (of class Type)
 - ✓ Invocations with return of a class

ASSIGNMENT

- `<variable> = <expression>`
- The result of evaluating the expression is copied into the contents of the variable.
 - ✓ The variable and expression must be of a compatible type.
- The assignment of primitive values is "different" to the assignment of references.

ASSIGNING PRIMITIVE VALUES

```
i1 = 5  
i2 = i1  
i2 = 0
```

- i1 and i2, are two variables with primitive values.
 - ✓ i1 has a value that is copied in i2
 - ✓ From there on: i1 and i2 store equal values but are still 2 different variables
 - ✓ If I modify one of them, the other one is unaltered.

ASSIGNING OBJECT REFERENCE

```
o1 = Thing(5)
o2 = o1
o2.setValue(0)
```

- O1 and o2 are two variables that store a value.
 - ✓ o1 has a reference that is copied into o2
 - ✓ From there on: though o1 and o2 are different variables, the two reference the same object
 - ✓ If I modify the content of one, I'm not altering the second one, but any change to the referenced object, I'll see it if I access from the second one.

ASSIGNING OBJECT REFERENCE

```
a1 = [50, 100]
a2 = a1
a1[0] = 5
a2[1] = 10
```

- Lists are objects that store several items.
- a1 and a2 are reference variables that have a reference to the same list:
 - ✓ a1 has a reference that is copied into a2
 - ✓ a1[0] is the same as a2[0]
 - ✓ a1[1] is the same as a2[1]

ASSIGNING OBJECT REFERENCE

```
a1 = [50, 100]
a2 = [50, 100]
a1[0] = 5
a2[1] = 10
```

- Lists are objects that store several items.
- a1 and a2 are reference variables that have a reference to different lists:
 - ✓ a1[0] is NOT the same as a2[0]
 - ✓ a1[1] is NOT the same as a2[1]

ASSIGNING OBJECT REFERENCE

```
thing1 = Thing(1)
thing2 = Thing(2)
a1 = [];
a2 = [thing1, thing2]
A1.append(Thing(5))
a2[0] = a1[0];
a2[0].setValue(10);
```

- I have 2 object's lists. Each element of the list store a reference.
- a1 and a2 are reference variables to "Thing" lists.
 - ✓ a1[0] is a reference that is copied into a2[0]
 - ✓ From there on: a1[0] references same object as a2[0]

CONCLUSION (I)

- The assignment always copies in the contents of a variable a value.
- The only way to modify a variable is with an assignment.
- Modify the contents of a variable != Modify the state of an object
 - ✓ Value variable
 - ✓ Modifying saved data
 - ✓ Reference Variable
 - ✓ Modifying the referenced object

CONCLUSION (II)

- An object can be referenced by many variables.
 - ✓ Modifying the state implies that all references will see the modified object
- A primitive value can be stored in several variables.
 - ✓ Modify one of them does not modify the rest the same way

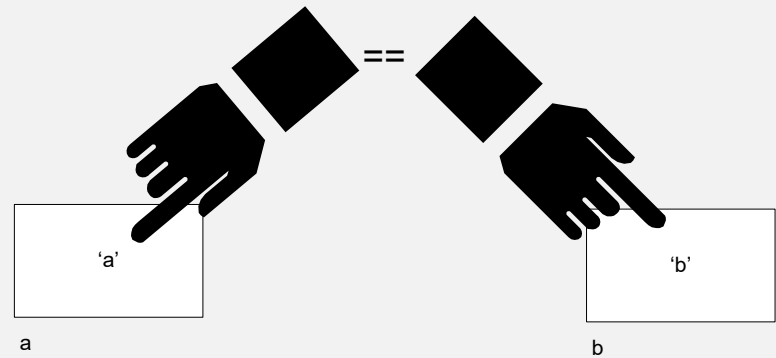
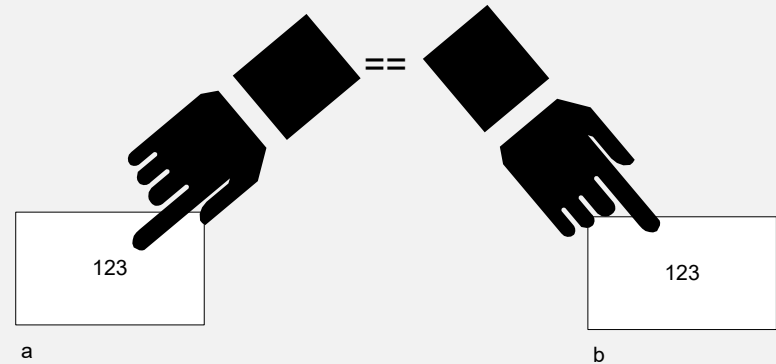
OPERATOR ==

- Operator == compares an expression on the left side with an expression on the right side.
- Comparison of primitive values is “different” to comparing references.
- Let’s see some examples...

OPERATOR ==

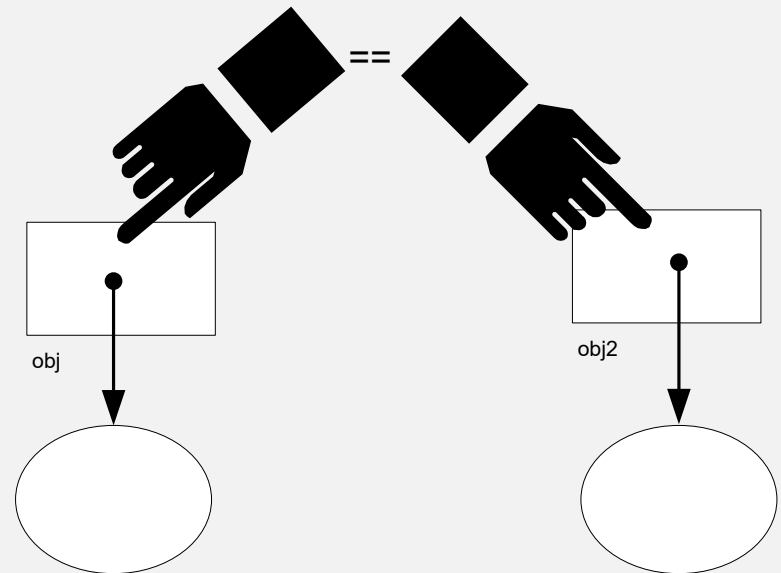
If a and b are variables of a primitive type its **contents are values**

- ✓ Values are compared.
- ✓ Comparison of equality



OPERATOR == (II)

- If a and b are variables of a primitive type its **contents are references**
 - ✓ References are compared or contents are compared.



OPERATOR == (III)

- Objects are not saved directly in the variables but a reference is saved to a place of memory where the objects are.
- Comparing variables regardless of their type has in common that we compare contents, but if we consider the type of the variable: that content can be a value or a reference.

OPERATOR == (IV)

- When you want to compare the equality of two objects, you have the following code fragment:

```
x1 = [10, 20, 30]
x2 = [10, 20, 30]
if x1 == x2:
    print("Yes")
else:
    print("No")
```

- What will be printed?
- Remember that Lists are objects

OPERATOR IS

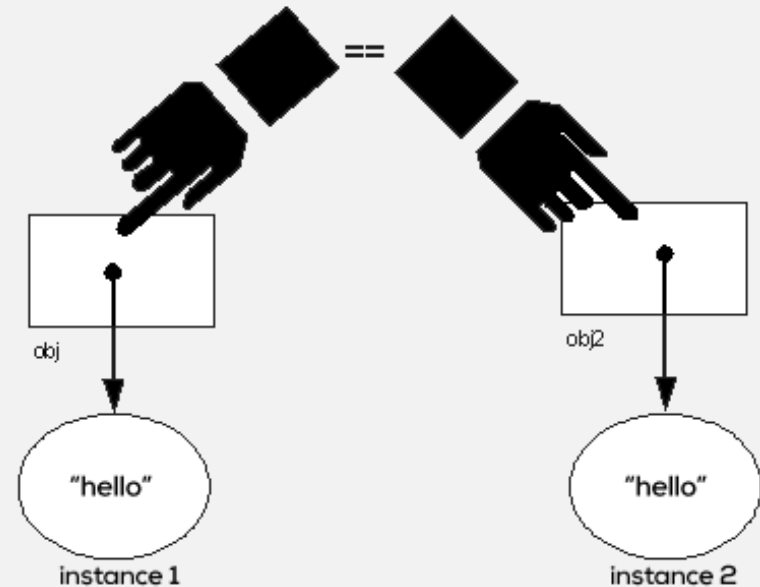
- Now we have the following

```
x1 = [10, 20, 30]
x2 = [10, 20, 30]
if x1 is x2:
    print("Yes")
else:
    print("No")
```

- What will be printed?
- Remember that Lists are objects

OPERATORS == AND IS (V)

- Since the references stored in s1 and s2 are different, they point to different objects
- Though the two objects are composed of the same data
 - ✓ They are two different objects, this is reflected in the fact that x1 and x2 will effectively refer to different objects.



OPERATOR == (VI)

- However, not all the classes we create have a built in ==. So we need to define it:

```
class MyClass:
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def __eq__(self, other):
        if not isinstance(other, MyClass):
            # don't attempt to compare against unrelated
types
            return NotImplemented

        return self.foo == other.foo and self.bar ==
other.bar
```

VARIABLES IN MEMORY

- So far we have represented variables conceptually as a box.
 - ✓ In reality, this is a little bit different.
- For a more accurate representation of the variables we have to get to know the RAM.
 - ✓ As the variables are "zones of the RAM" in which the data is stored.

VARIABLES IN MEMORY

- Memory is represented as a grid that is divided into cells
 - ✓ Each cell is numbered and it has the capacity to store one byte
- Variables occupy one or more adjacent cells (depends on the type of data they store).
 - ✓ In this cell they store the content
 - ✓ For now we'll assume that each value is stored in one cell.

MEMORY MANAGEMENT

- There are two parts of the memory
 - ✓ **Stack** where local variables and parameters are stored.
 - ✓ **Heap** where objects are stored.

MEMORY MANAGEMENT

- As the interpreter executes the code and finds itself with declaration of local variables or invokes a method and it has parameters automatically reserves memory zones of the Stack to save the values of those variables.
- When we create an object we reserve a zone of memory of the Heap and we keep there that object.
- We'll see why this is useful, in the future.

CONCLUSION

- Reference variables are compared using == or is.
 - ✓ A method is usually defined to compare the content of an object with another one
 - ✓ What is usually done is to overwrite the “==” that receives as a parameter another Object.
- **Golden Rule** (for reference variables):
 - ✓ Variable content != Object content

PARAMETER PASSING

- We have already seen that the arguments end up being expressions of the type corresponding to the parameters that are expected
 - ✓ In one side (the arguments) I have expressions
 - ✓ On the other (parameters) I have variables.
 - ✓ There is an assignment of results of the expressions that play as arguments to the variables that are the parameters.
- Parameters are variables, so they are created and assume a value when that method is invoked and then at the end they die.

PARAMETER PASSING

- If the parameters are variables of value
 - ✓ The modifications alter only the content of said variable and nothing else.
 - ✓ Changes will not be reflected after method execution.

PARAMETER PASSING

- If the parameters are reference variables, we can:
 - ✓ Or modify the contents of that variable, that is, modify that referenced object.
 - ✓ Or we can modify the object that references that variable by accessing its methods (messages) or attributes.

PARAMETER PASSING

- So, our most important conclusion is:
 - ✓ If you modify the content of a parameter, we are not altering the contents of any variable outside the method.
 - ✓ But if we modify the state of an object referenced by the parameter then the changes will be final from there onwards.

OVERLOADING (I)

- In Python you can define a method in such a way that there are multiple ways to call it.
- Given a single method or function, we can specify the number of parameters ourself.
- Depending on the function definition, it can be called with zero, one, two or more parameters.

OVERLOADING (II)

```
#!/usr/bin/env python

class Human:

    def sayHello(self, name=None):

        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')

# Create instance
obj = Human()

# Call the method
obj.sayHello()

# Call the method with a parameter
obj.sayHello('Emerson')
```

OVERLOADING (III)

- Overloading is useful to avoid thinking of new names when the method performs the same action even though it receives different parameters.
- This concept also serves for constructors, so we can optionally provide different ways to initialize the state of an object.

EXERCISING

PRACTISE 3

THE END

QUESTIONS?

ComIT

OOP IV

THE TOPICS

- Contexts
- Identifiers
 - ✓ Scope
- OO terminology
 - ✓ Object collaboration
 - ✓ Messages

CONTEXTS

- Let's introduce the concept of context
 - ✓ It has to do with the "place" where sentences are written
 - ✓ Instance Contexts
 - ✓ Class Contexts
 - ✓ Let's see an example...

LET'S START

```
class Thing
    def __init__(self)
        self.__num=0

    def assignOne(self)
        Self.__num = 1
```

A small class named "Thing" with a "num" attribute and a method to assign that attribute to one.

QUESTION

```
class Thing
    def __init__(self)
        self.__num=0

    def assignOne(self)
        self.__num = 1

t1 = Thing()
t1.__num=1
t1.assignOne()
```

Can one ask why the red line fails and the green one is perfect?

ANSWER

- The "num" variable is an attribute and therefore belongs to an object.
- If we want to access "num", we need to indicate the "num" of which object we want to access.
- Let's see why it DOES NOT fail to assignOne and then why it fails when we try to access num directly.

WHY DOESN'T FAIL IN ASSIGN ONE

- Within the method "assignOne" we are in the context of an object
 - ✓ And to invoke it, it is required to specify an object
 - ✓ To which we request to carry out this method.
 - ✓ If we write "num" we are using the num attribute of the object whose method assignOne is being invoked
 - ✓ In the method I can use “self” to access a private attribute
 - ✓ Instance method

WHY DOES IT FAIL IN THE MAIN PROGRAM

- Within the "main program" we are not in the context of an object
- When we talk of "num" it is not understood what it is
 - ✓ I have to specify the "num" of which object I want to access (that is also private to the object), using an explicit reference.
- That is why using the "self" in the main program does not make sense, since I am not in the context of some object, so "self" does not refer to anything.

TO SUM UP

- To invoke methods or use attributes of an object, depending on the context where we are, we have to specify, or not, the object we are talking about. Whether or not we need an instance.

TO SUM UP

- In a “generic” context to access the members of an object we have to explain the reference to it. (Explicit invocation).
- In an object context, if we do not specify anything, the members of those objects are accessed (implicit invocation).

TO SUM UP

- An instance method or attribute is associated with an object, since to be executed or accessed requires a specific object.
- Class attributes are not associated with an object.

IDENTIFIERS: SCOPE RULES

- An identifier is a symbolic name that we give to a program element (variable, class, etc.), which we use to refer to it and use it throughout the program.

EXAMPLES

- Specifically an identifier is the name given to...
 - ✓ A class
 - ✓ An attribute (class or instance)
 - ✓ A method (class or instance)
 - ✓ Local Variable
 - ✓ A Parameter

SCOPE

The scope of an identifier is the portion of the code in which said identifier may be "seen" and, therefore, used.

SCOPE OF IDENTIFIERS

- Local Variables and Parameters
 - ✓ They can be accessed from the method where they were declared on.
- Object Members
 - ✓ They can be accessed while having a reference to that object.
 - ✓ Accessibility is altered using the access modifiers.

CONCEPTS OF MESSAGE AND COLLABORATION

MESSAGES

- Objects interact with each other and do so by sending messages or requests.
- Objects react to messages by activating the behavior that has been defined.

MESSAGE COMPONENTS

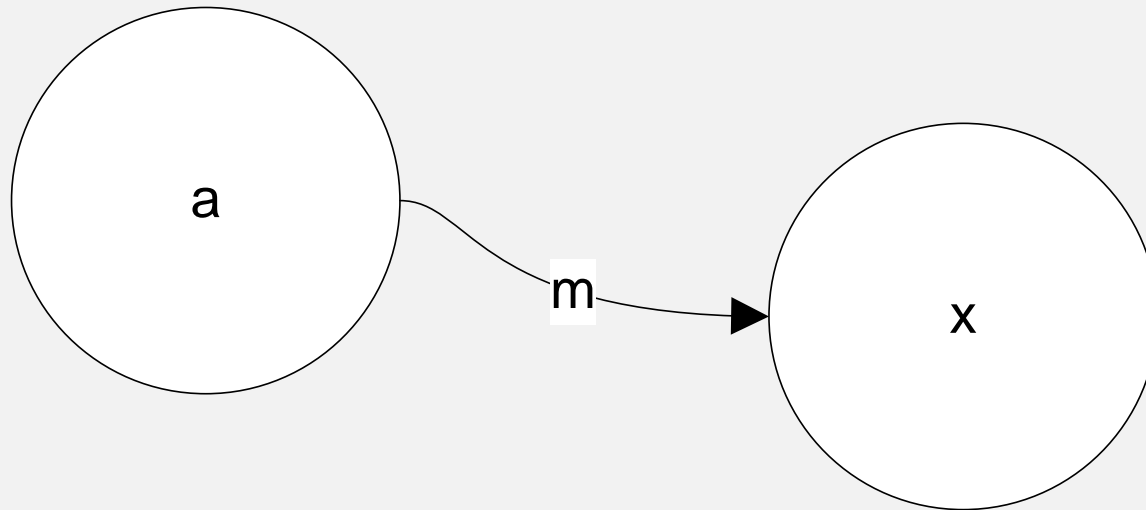
- We identify 3 components
 - ✓ **Sender Object**
 - ✓ Requests the service
 - ✓ Composes the message
 - ✓ **Message**
 - ✓ **Receiver Object**
 - ✓ Executer of the service
 - ✓ Receives the message

MESSAGES

- A message translates as an invocation of a method of a given object.
 - ✓ Executing code contained in an instance method of the object receiving the message

MESSAGES

- Properly Speaking
 - ✓ Invoking a method "m" of an object "x" is equivalent to sending the message "m" to the object "x"



MESSAGES EXAMPLES

- When John turns off the television from his house to go to sleep.
 - ✓ The object John (of the class Person) is sending the message to the TV object (TV class)

COLLABORATION

- From the delicate interaction between objects we obtain the resolution to the problem that the program tries to solve.
- A program is a set of objects **collaborating** with each other to accomplish a goal.
- We can see the system as a big object
- Composed of smaller ones

DESIGN TECHNIQUES

- The goal of our study as we learn the rules of languages will be how to orchestrate these relationships between objects to build an application that solves the problem efficiently
 - ✓ Using an appropriate class design and object modeling
 - ✓ ***This takes time and practice***

COLLABORATION IN THE CODE

```
class Person

    def turnOn(self, tv)
        Tv.turnOn()
```

- **turnOn** is an instance method that runs in an object context, Person in this case.
- **An instance of Person** is going to send the message to object **tv**, sent as a parameter, so that it turns on.

COLLABORATION IN THE CODE

```
class Person  
  
    def turnOn(self, tv)  
        tv.turnOn()
```

- Message `turnOn` is received by **referenced object**
- Message `turnOn` is sent by object whose method `turnOn` is executed
 - ✓ I need to know the context in which I'm working on

COLLABORATION IN THE CODE

```
class Person  
  
    def turnOn(self, tv)  
        tv.turnOn()
```

```
tv = Television()  
per = Person( "john" )  
per.turnOn(tv)
```

- Message **turnOn** is sent to **referenced object** (**"tv"**)
- Message **turnOn** is sent by the **object whose method turnOn is executed**
- What is the name of the person who sends the message to the TV?

COLLABORATION IN THE CODE

```
class Person
```

```
    def turnOn(self, tv)
        tv.turnOn()
```

```
tv = Television()
per = Person( "john" )
per.turnOn(tv)
```

- Message **turnOn** is sent to **referenced object** (“per”)
- Message **turnOn** is requested by...¿?
 - ✓ I’m not in an object context
 - ✓ I’m in a **static context**
 - ✓ “we” (or “god” or “nobody”)
 - ✓ We indicated to send the **message** to the **object**.

COLLABORATION IN THE CODE

- To know who originates the message you have to know in what context I am
 - ✓ Context of instance or object
 - ✓ It is an object of the class who requests
 - ✓ Static Context
 - ✓ “We” are who request

EXERCISING

PRACTICE

THE END

QUESTIONS?

ComIT

00P V

THE TOPICS

- Encapsulation
- Interfaces
 - ✓ Concept
- Garbage Collector
 - ✓ Destruction
 - ✓ Finalize
- More Modifiers

ENCAPSULATION

- Encapsulation is one of the main benefits of OOP
- "Wrap everything in one place"

RULE TO ENCAPSULATE

- **Encapsulation** consist of:
 1. Keep attributes protected
 - ✓ Using access modifiers
 2. Public methods to manipulate the state consistently

MEANING OF ENCAPSULATE

- Hide the attributes of an object, promoting indirect access to them using methods.
- Think of an object as a black box that does things but we do not know how it does them.

OBJECT AS A BLACK BOX

- Each object is isolated from the outside and only interacts through methods.
- No matter how the methods are implemented, as long as they do what they have to do.

PROS

- Eliminates the need to know the underlying complexity of each object
 - ✓ Ignore the code in a method
- It prevents you from corrupting the state of an object.
- It allows to significantly improve the quality of the code
 - ✓ Flexibility and Extensibility

ENCAPSULATION

Let's see with an example how to properly apply the mechanism of encapsulation to greatly improve the quality of the code.

```
class Test
    def __init__(self):
        self.grade=0
```

```
par = Test()
par.grade = 5
```

- **Not** encapsulated

PROBLEMS

- Can I assign the value 11 to the grade attribute?
- The only way to protect the state is to make the attribute private and a public method that modifies the state in a controlled way
 - ✓ Encapsulate

NOT CORRUPTIBLE OBJECT

```
class Test
    def __init__(self):
        self.__grade=0

    def setGrade(self,grade):
        if (grade>=0 and grade<=10):
            self.__grade=grade

    def getGrade():
        print (self.__grade)

par = Test()
par.setGrade(5)
```

- We added a condition to prevent invalid states

IMPROVING QUALITY

- What about improving the quality of my code?
 - ✓ Encapsulation is a way to start
- Let's continue...

IMPROVING QUALITY

- Let's assume the object is not encapsulated
 - ✓ Public attribute grade
- We want to attack the potential problem of assigning an invalid value
- Then we contemplate the validation each time we assign a value

IMPROVING QUALITY

```
par = Test()

# a lot of code

if (value >= 0 and value <= 10):
    par.grade = value

# a lot of code

if (value >= 0 and value <= 10):
    par.grade = value
```

- If we change the validation, we have to change the code in each part of the program where it is

IMPROVING QUALITY

- It's a big issue
 - ✓ Making changes is more error-prone simply by having to change it on several sides
 - ✓ Especially in large projects where you can repeat the same operation in different parts of the code along different files

IMPROVING QUALITY

- A good encapsulation brings the benefit of concentrating an operation in one place.
 - ✓ **Place = Method**
 - ✓ Avoiding replication in the program

IMPROVING QUALITY

```
par = Test()  
# a lot of code  
par.setGrade(value)  
# a lot of code  
par.setGrade(value)
```

- If we change the validation, we change code **only in the method**

GOOD ENCAPSULATION

- Each class has to include the logic corresponding to it.
 - ✓ Called responsibility
- This conditions a good design
 - ✓ What happens if the condition **grade** between 0 and 10 is not for every object
 - ✓ Some can now be between 0 and 100
 - ✓ Then the above code is no longer valid
 - ✓ Class design tends to be complicated

ENUM TYPES

ENUM TYPES

VARIABLES WITH KNOWN VALUES

- What happens when we want a variable to store a type of data which we know that there is only a limited and countable amount of possible values?

VARIABLES WITH KNOWN VALUES

- E.g.:
 - ✓ Days of the week
 - ✓ 7 possible values
 - ✓ Stores dollar bills
 - ✓ 5 possible values
 - ✓ Students grade
 - ✓ 10 possible values
- ***How do we manage these variables?***

PROPOSAL

- Use values of a primitive data type to represent the data in question.
- For example:
 - ✓ To store the day of the week use a variable of type **int**
 - ✓ Knowing that there is a correspondence between the numerical value and what it represents, in this case, the day of the week.
 - ✓ 1 → Monday
 - ✓ 2 → Tuesday
 - ✓ ...
 - ✓ 7 → Sunday

PROPOSAL

- Pros
 - ✓ Simple
- Cons
 - ✓ Tedious
 - ✓ It is necessary to remember the correspondence between value of the primitive data type and what it represents.
 - ✓ Inconsistencies
 - ✓ Day = 20;
 - ✓ Inflexibility
 - ✓ If I modify the representative values I have to alter all the parts of the code where I used those values

INFLEXIBILITY

- Monday is 1

```
day = 1
if (day == 1):
    print("It's Monday")
```

- If I change the current representative value of Monday (which is 1) by 0, then I must change line by line where it was used:

```
day = 0
if (day == 0):
    print ("It's Monday")
```

PROPOSAL (II)

- Use an existing primitive data type to store a value representative of the data.
- Define a class with public static constants to make assignments and manipulation more readable.

PROPOSAL (II)

- To store the day of the week using a variable of type int.
- I define a class to write in it constants that have values of type int that represent the days of the week.

PROPOSAL (II)

```
class WeekDay
```

```
    MONDAY = 1  
    TUESDAY = 2  
    WEDNESDAY = 3  
    THURSDAY = 4  
    FRIDAY = 5  
    SATURDAY = 6  
    SUNDAY = 7
```

- Public to be accessible from anywhere
- Of Class so it belongs to all the object WeekDay
- Constant so they can not be modified
- They are cute since they are understood even by someone with no experience in programming

PROPOSAL (II)

- We can work like this:

```
day = Weekday()  
d=day.MONDAY  
d=day.THURSDAY  
if (d == day.FRIDAY)  
    ...
```

- While assigning and comparing with values defined in the WeekDay class we are working in a "safe mode".
 - ✓ Without inconsistencies

PROPOSAL (II)

- Pros
 - ✓ Simple
 - ✓ We just have to use an existing data type and a class that lists the possible values
 - ✓ Flexibility
 - ✓ If I modify the representative values I only modify the values of the constants defined in the class
- Cons
 - ✓ Inconsistency
 - ✓ `d = 20;`

PROPOSAL

- We saw two proposals....
- We can apply the encapsulation mechanism by wrapping that variable in an object
- but there is a much simpler, practical and effective way for these problems.

ENUM TYPES

- We can define a new data type by enumerating (exhaustively) all the possible values that a variable of that type can assume.

ENUM TYPES

- Are defined in the same way as a class

```
>>> from enum import Enum
>>> class Color(Enum):
...     red = 1
...     green = 2
...     blue = 3
... 
```

EXAMPLE

- Defining an Enum is similar to a Class definition:

```
from enum import Enum  
class Weekday(Enum):
```

```
    MONDAY=1
```

```
    TUESDAY=2
```

```
    WEDNESDAY=3
```

```
    THURSDAY=4
```

```
    FRIDAY=5
```

```
    SATURDAY=6
```

```
    SUNDAY=7
```

EXAMPLE

- To assign a value we use “=”

✓ `day = Weekday.MONDAY`

- To compare, we use “==”

✓ `day == Weekday.FRIDAY`

EXAMPLE

- **Comparisons against non-enumeration values will always compare not equal**

✗ `day == 3`

✗ `day == 'c'`

- ✓ A validity check is added during compilation which helps to substantially reduce errors.

ENUM TYPES

- Pros
 - ✓ Simple
 - ✓ I only have to know the name of the type
 - ✓ Hiding the implementation
 - ✓ Virtual Machine chooses the value for the data
 - ✓ Consistency
 - ✓ We can't assign invalid valued.
- Cons
 - ✓ It is not always easy to determine an enum

ENUM TYPES

- In fact, they are a special kind of Class in which we define beforehand the "counted" possible objects, which are known by the name of elements.

ENUM TYPES

- We can add methods and attributes that will be specific to the type.
- Each particular element (MONDAY, TUESDAY, MAN, WOMAN, etc.) is an "instance" of this special type of class.
- These are pre-built objects by the compiler
 - ✓ You can not invoke your constructor explicitly to construct a new element.

ENUM TYPES WITH MEMBERS

```
>>> class Mood(Enum):
...     funky = 1
...     happy = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.happy
```

ENUM TYPES WITH MEMBERS

- And we can access the members of an element through the identifiers of the enum, just as we do with a reference to an object

```
>>> Mood.favorite_mood()
<Mood.happy: 3>
>>> Mood.happy.describe()
('happy', 3)
>>> str(Mood.funky)
'my custom str! 1'
```

INTERFACES

A WAY OF LIFE

THE IMPORTANCE OF INTERFACES

- Though in Python there are no Interface Objects like in Java, it is important to understand the “interface” concept as it applies to the interaction between classes, modules, subsystems and systems.
- The interface concept will be used throughout your career to use APIs, or to develop new systems that plug with others.

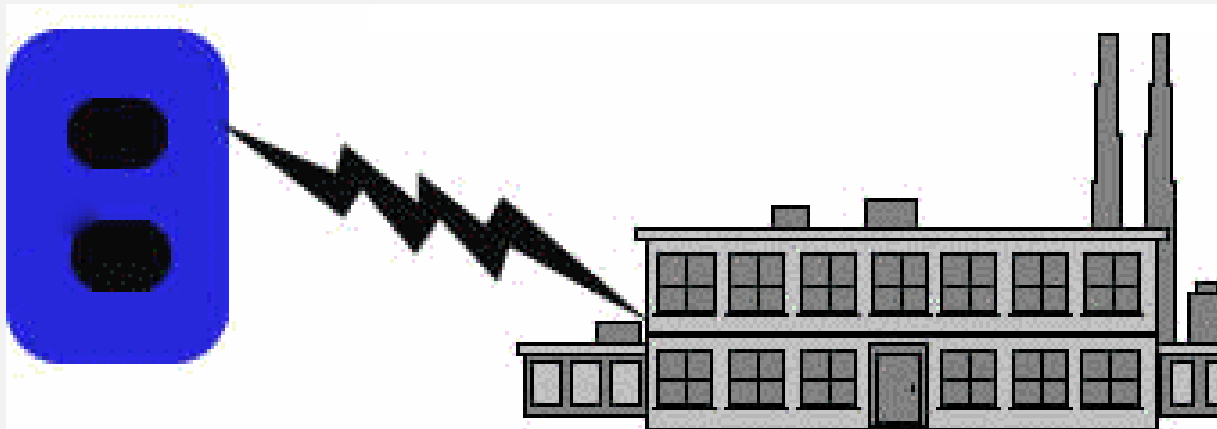
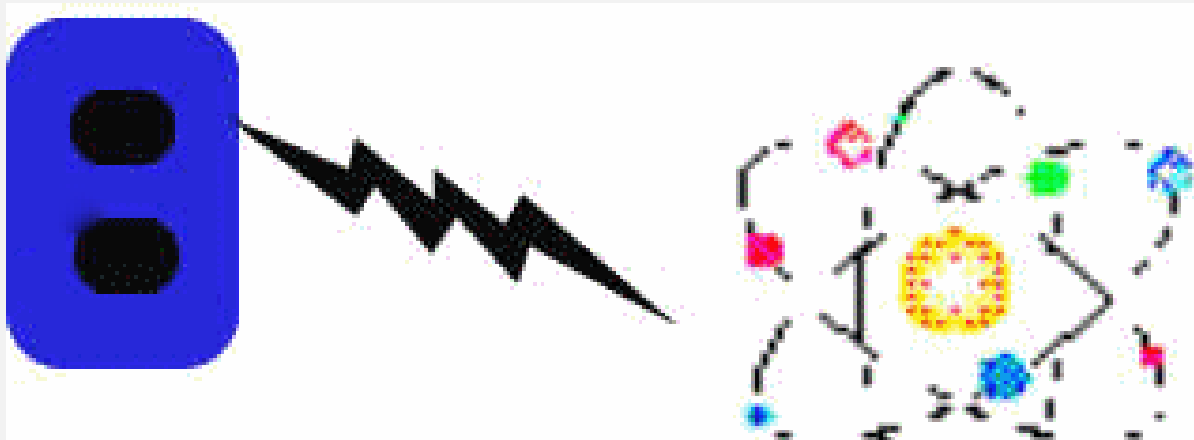
INTERFACES

- It is something that is between two parts that maintain a contact.
- A point where two (or more) independent systems interact.
- Interfaces are best explained with real-life examples.

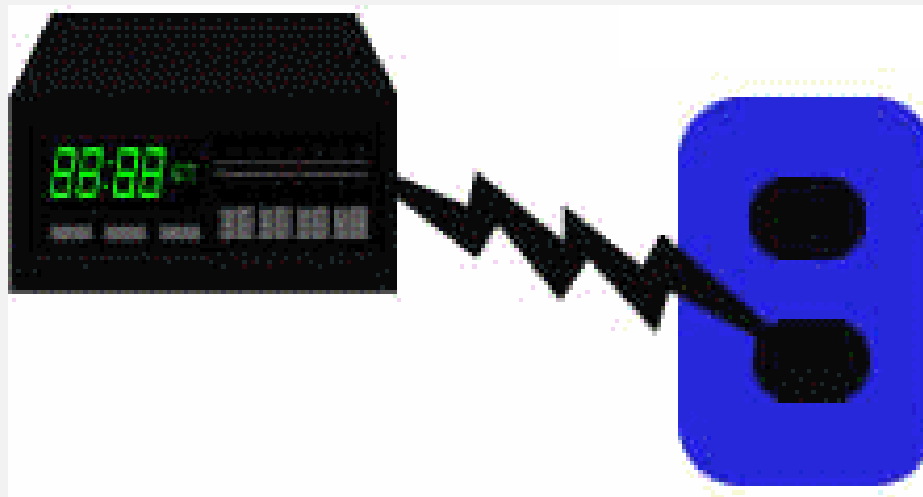
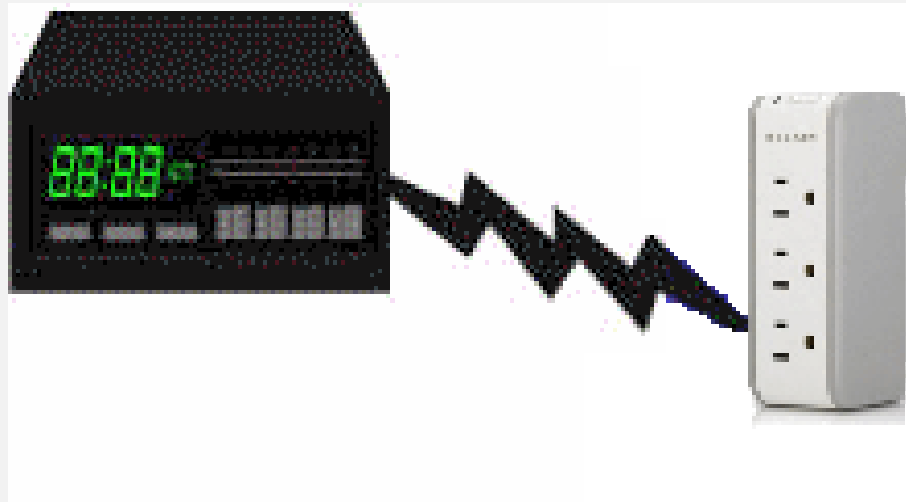
INTERFACES

- E.g.: a power plug
 - ✓ Need to connect to an electricity provider
- Does the power plug care about where the electricity comes from?
 - ✓ NO
 - ✓ What matters is that it complies with a standard of connection to be able to connect with a power supplier.
 - ✓ It doesn't matter where or how electricity is produced

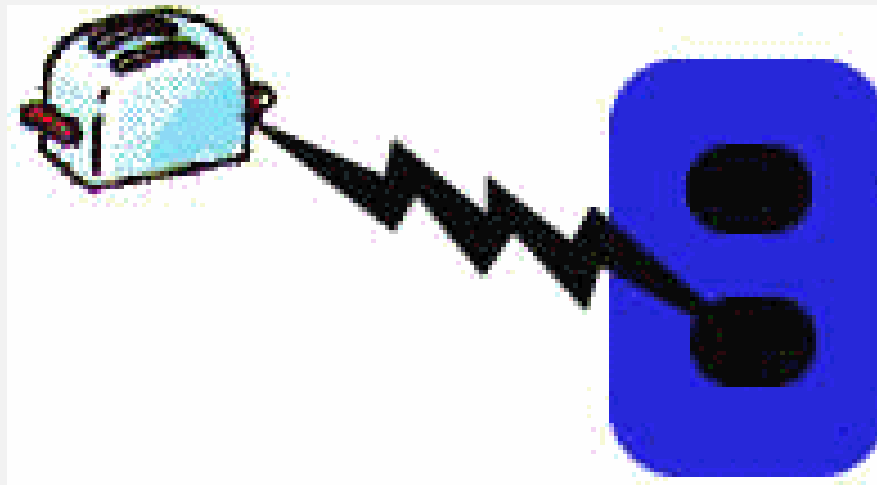
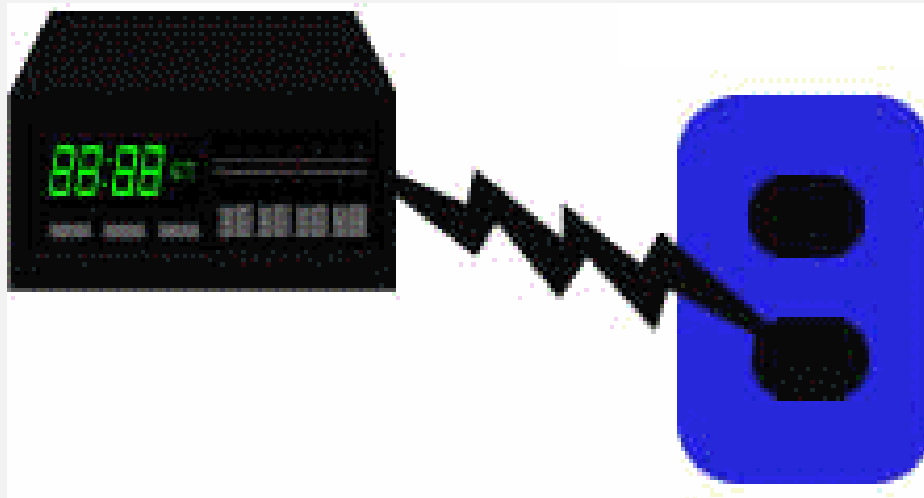
INTERFACES



INTERFACES



INTERFACES



INTERFACES IN REAL WORLD

- A proper definition
 - ✓ The interface is the functional connection between two independent devices or systems
 - ✓ Sometimes it's physical
 - ✓ Like previous examples
 - ✓ Sometimes it's logical
 - ✓ Like two people talking

INTERFACES IN AN OBJECTS WORLD

- The interface is the "visible part" of an object with which another object can interact.
 - ✓ Public methods reflect the interface of the objects of a class
 - ✓ When an object sends a message to another, it does so through some method specified in that interface.

INTERFACE AND BEHAVIOR

- What's the difference between interface and behavior?
 - ✓ Behavior is a more "complete" definition than interface
 - ✓ Behavior refers to the functionalities that an object presents, both in terms of "what it does" and "how it does it".
 - ✓ Interface is "what it does"
 - ✓ In terms of code:
 - ✓ Behavior = Method Declaration {+ Definition}
 - ✓ Interface = Method Declaration

INTERFACES

- Objects of very different classes can have the same interface.
 - ✓ It means that they share some behavior, even if you implement it differently

INTERFACES

- It allows to imitate the component industry, where one part is hooked with another
- Without knowing the internal specifications
- Knowing the interface

INTERFACES IN INDUSTRY

- As in the assembly of a car, where the factory buys the audio equipment
 - ✓ The car manufacturers are not interested in how the audio equipment works inside
 - ✓ They are only interested in complying with a certain interface to be able to assemble it to the car so that it works.

INTERFACES IN SOFTWARE

- Interfaces allow us to create interchangeable or uncoupled software modules.
- So that one module can use another with the condition that it implements a specified interface.
- Let's look at a "real" example applied to the software industry...

A SOFT STORY

- In a software development company a "Boss" asked 3 programmers to develop a class that contains the artificial intelligence for a chess game.
- This boss wanted to use this class.

INTERFACES - CONT.

- In this way the interfaces are a mechanism to declare some methods that must be implemented by the classes
 - ✓ Establishing a standard
- It is said that the classes that provide a concrete code for those methods implement this interface.

INTERFACES - CONT.

- An interface **defines**
 - ✓ What the object does
 - ✓ And not how it does it
 - ✓ It standarizes

INTERFACE AS A CONTRACT

- It starts from assuming a contract between two parties.
- The implementing party.
 - ✓ Must provide code that respects the specification
 - ✓ Methods names, parameters and return types
- The party who uses it.
 - ✓ Use the code according to specification
- Specs is what both know
 - ✓ Preconditions
 - ✓ How this operation is invoked (Names, parameters and return types)
 - ✓ Postconditions

WHO SAYS IT'S EASY?

- The concept of interface, although we use it on our daily life, it's difficult to take it to OOP.

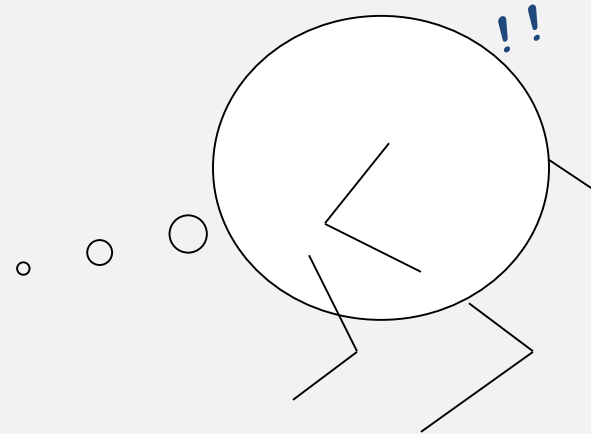


EXERCISING

EXERCISING

GARBAGE COLLECTOR AND OBJECT FINALIZATION

Forgotten object



GARBAGE COLLECTOR

- In Python we can create objects without worrying about destroying them explicitly.
- The Python runtime environment deletes objects when they are determined to be no longer in use.
 - ✓ Garbage collection.

GARBAGE COLLECTOR

- An object is eligible for garbage collection when there are no more references to that object.
- The references that remain in a variable disappear when the variable is outside its scope.
- Or, you can drop the reference to an object by establishing the special value `None`.
 - ✓ `varRef = None`

OBJECT DESTRUCTION

- In Python it is normal for several reference variables to point to the same object, so a counter is internally carried for how many references there are on each object.

```
ref1 = Thing()  
ref3 = ref2 = ref1
```

OBJECT DESTRUCTION

- When the number of references is zero, the object will be deleted in the next execution of the garbage collector.
 - ✓ We don't know when the garbage collector is executed.
 - ✓ It is invoked only when the memory is exhausted and space needs to be freed to allocate new objects.

CALLING GARBAGE COLLECTOR

- There is a way to invoke the garbage collector explicitly ...
- When you really need to have memory released

```
import gc  
gc.collect()
```

THE END

QUESTIONS?

ComIT

OOP VI

THE TOPICS

- Modules
- Error Management
 - ✓ Conservative approach
 - ✓ Optimist approach
 - ✓ Call Stack
- APIs

THE NEED TO ORGANIZE

- In a small project, we usually include all Python code in a single file.
- However, if our project gets bigger, and the number of classes is increasing, putting everything in the same file might be a nightmare.

MODULES

- How do we solve this issue?
 - ✓ **Creating different files**
- In Python, instead of calling it just files we call them **modules**.
- A module allows us to logically organize our Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

MODULES

- Using modules we can group classes in conceptual units (file) according to:
 - ✓ Functionality
 - ✓ Conceptual category

WHERE ARE MY MODULES?

- When you import a module, the Python interpreter searches for the module in the following sequences –
 - ✓ The current directory.
 - ✓ If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
 - ✓ If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.
- The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

USING MY MODULES

- There are different statement to use our modules:
 - ✓ `import module1[, module2[,... moduleN]`
 - ✓ When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
 - ✓ `from modname import name1[, name2[, ... nameN]]`
 - ✓ Lets you import specific attributes from a module into the current namespace.
 - ✓ `from modname import *`
 - ✓ This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly

NAMESPACES AND SCOPING

- Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values).
- A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- Therefore, in order to assign a value to a global variable within a function, you must first use the `global` statement.
- The statement *`global VarName`* tells Python that `VarName` is a global variable. Python stops searching the local namespace for the variable.

MODULES - PROS

- **Enable better organization**
 - ✓ Software is easier to maintain and enables better collaboration
- Avoid class name collision.
 - ✓ I can have classes with the same name in different modules

ERROR HANDLING

IN RUNTIME

ERROR HANDLING

- What kind of errors?
 - ✓ Not grammatical or logical errors
 - ✓ Runtime Errors
 - ✓ For invalid data entry
 - ✓ Poorly managed processing flow
 - ✓ Internal OS issues
 - ✓ **Errors that alter the normal program execution flow.**

ERROR HANDLING

- Handling errors adds algorithmic difficulty when writing code.
- Two approaches to address errors
 - ✓ **Conservative** Approach
 - ✓ **Optimist** Approach

CONSERVATIVE APPROACH

- First I ask, then I do
 - ✓ First, I verify conditions, and if everything is OK then I perform the operation.

CONSERVATIVE APPROACH

- Number division
 - ✓ Condition: `divider != 0`

```
if divider != 0:  
    res = dividend / divider
```

- What do we do if condition = false?
 - ✓ Reject: Print error message
 - ✓ Retry: retry operation

CONSERVATIVE APPROACH - PROBLEMS

- Code that performs the operation coupled with the code in charge of dealing with possible problems.
 - ✓ Poor code readability
 - ✓ Poor portability
 - ✓ The class can become non-reusable if it requires different error handling in another context

OPTIMIST **APPROACH**

- First, I try, then I ask
 - ✓ Processes or operations defined in **methods**
 - ✓ We must communicate if the operations was **performed or not**
 - ✓ Extra Parameters
 - ✓ Operation code
 - ✓ Operation Result
 - ✓ Operation Method and Verification Method

OPTIMIST APPROACH – OPERATION CODE

```
class Calculator
    def divide(self, dividend, divider, operationCode):
        res = 0
        operationCode.setValue(0) # initialize in 0 0
        if dividend != 0:
            res = (dividend / divider)
        else
            operationCode.setValue(1) # if error, assign 1

        return res # return result, only vali if cod == 1
```

- **Usage**

```
res = calculator.divide(2,3,cod)
if cod.getValue() == 0: # 0 operation code OK
    print("Result is: ", res)
```

OPTIMIST APPROACH – OPERATION RESULT

```
class Calculator
    def divide(self, dividend, divider, result):
        cod = 0 # initialize in 0
        if dividend != 0:
            result.setValue(dividend / divider) # save div
        else
            cod = 1 # code in 1 if error

        return cod # return code
```

- **Usage**

```
cod = calculator.divide(2,3,res) # First I do
if (cod == 0): # THEN I ASK if code == 0
    print("Result is: ", res.getValue())
```

OPTIMIST APPROACH

- It is not completely different from conservative approach
 - ✓ There is always a validation
 - ✓ Inside the method
 - ✓ We check if the operation was performed or not
 - ✓ Outside the method
- It is an improvement.

OPTIMIST APPROACH - PROBLEMS

- Using methods with "altered" headers decrease the neatness of the code and the elegance of the solution.
- It is necessary to know greater specifications to use methods.
- The optimist approach is far from being a silver bullet.

PROBLEMS WITH BOTH APPROACHES

- We have to handle
 - ✓ Operation code values
 - ✓ Verifying errors becomes an "if-else" cascade.
- Is there a better way?

ROAD TO A BETTER PLACE

- Python provides us with a more "clean" way to handle problems that can occur at runtime.
- **Exception management.**

EXCEPTION MANAGEMENT

- The goal is to write sentences to solve the problem without "thinking" about alternative flows
- Considering each alternative flow in another program location
 - ✓ Separating regular code from error handling.
- What is an exception?

WHAT IS AN EXCEPTION?

- An exception is an unexpected condition that interrupts the normal operation of the program.
 - ✓ Things that should not happen but happen
 - ✓ Generates an alternative flow

EXCEPTIONS

- Let's suppose

```
removeElements(list):
```

```
...
```

- ✓ What happens if you pass an empty list as an argument?
- ✓ What if we pass None?
- What if we one to access a list with a wrong Index?
- Two situations that may produce an alternative flow.

EXCEPTION MANAGEMENT

- Mechanism to interrupt the normal execution flow of a method, jumping to a specified portion of code to manage the problem
- There must be a connection between normal flow and articulated alternative flow through the exceptions

EXCEPTION MANAGEMENT

- To understand the mechanism of exception management we need to know a little bit more about invocation of methods

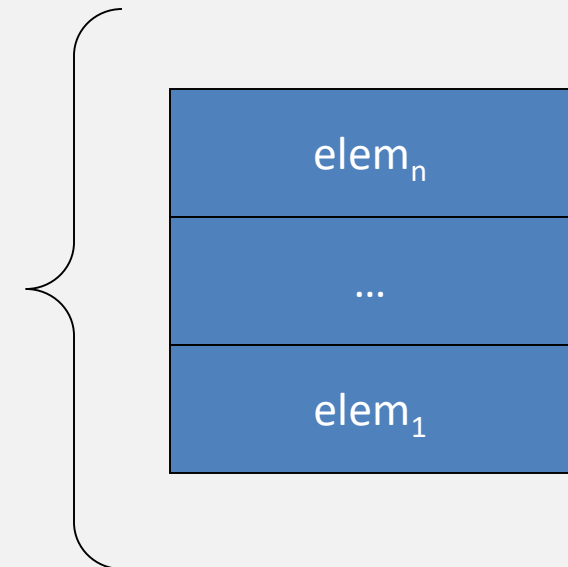
INVOCATION METHODS

- Methods run nested.
 - ✓ A method is always executed inside another one
 - ✓ Invoking Module (method from which it is invoked)
 - ✓ Invoked Module (invoked method)
- We start at main ()
 - ✓ Level “0”
- There is a data structure that is in charge of controlling the methods execution
 - ✓ Managing the flow

CALL STACK

- It is the “call stack”
- Each stack element (activation record) stores information about the methods being executed
- Let's see how it works...

“Call stack”

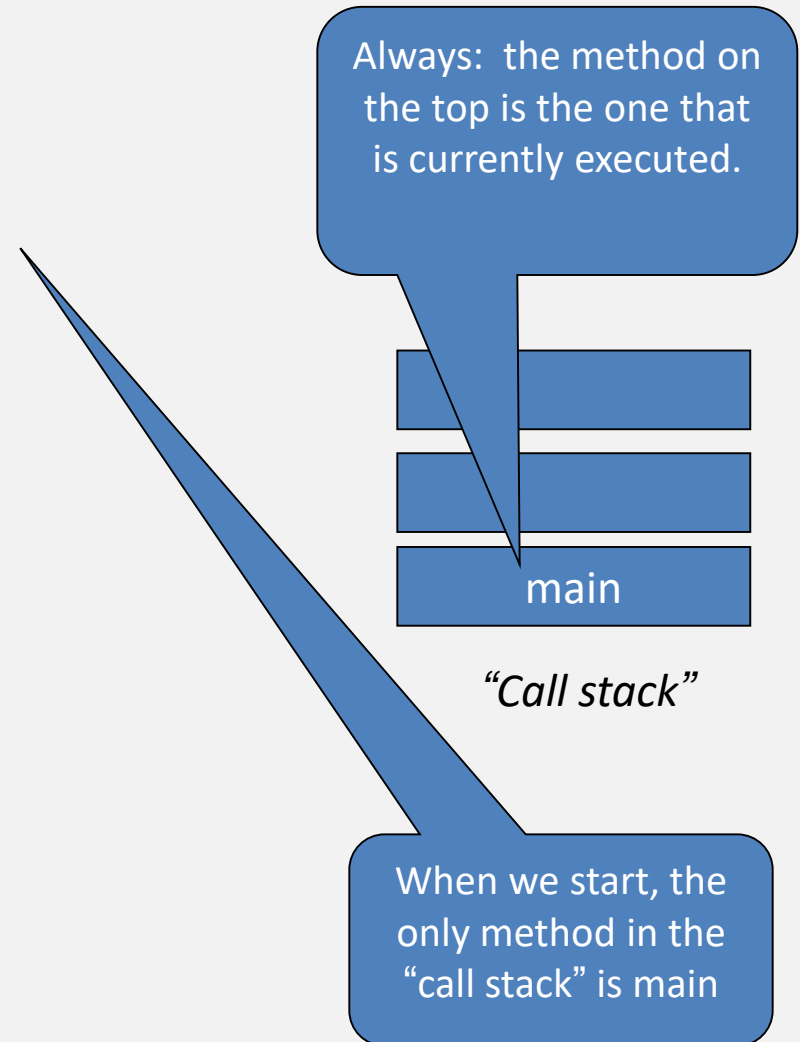


CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```



CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```

Methods are stack in
the order they are
invoked

printAbsdiv

main



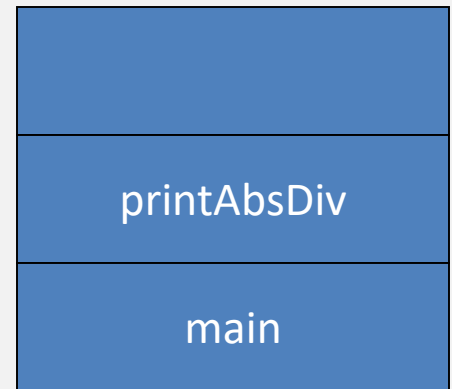
CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```

- Next invocation?

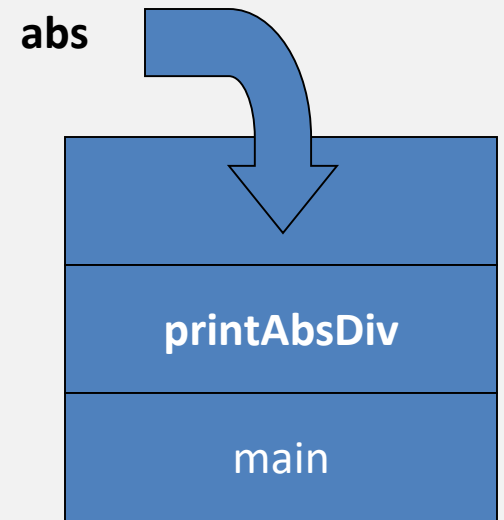


CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```



CALL STACK

```
def abs(a):  
    return (a < 0) ? -a : a;
```

abs

printAbsDiv

main

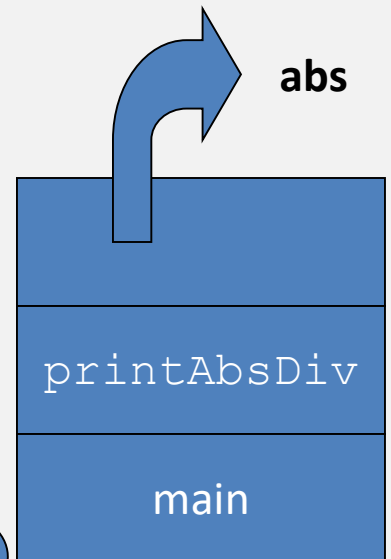
CALL STACK

```
def abs(a):  
    return (a < 0) ? -a : a;
```

A method ends when it finishes or **return**

When the method ends, it gets off the stack.

How do I continue?



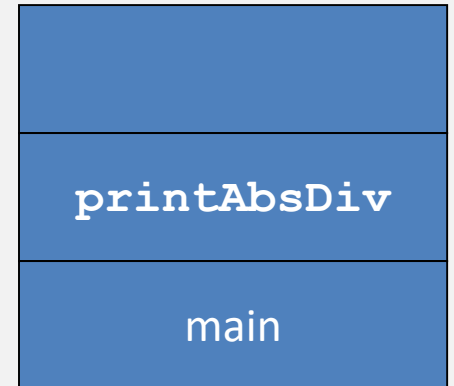
CALL STACK

Main Program

```
a = 2  
int b = 5  
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):  
    res = 0  
    res = a / abs(b)  
    print(res)
```

Continue here



The execution continues with the method at the top of the stack.

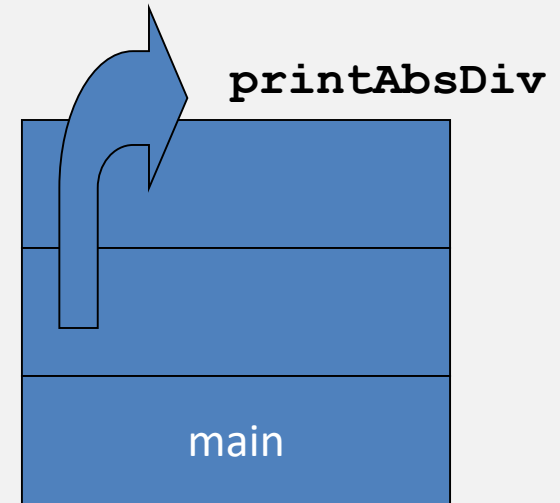
- ✓ It resumes at the point where it invoked the other method.

CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```



We skipped this call

End and main

CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```



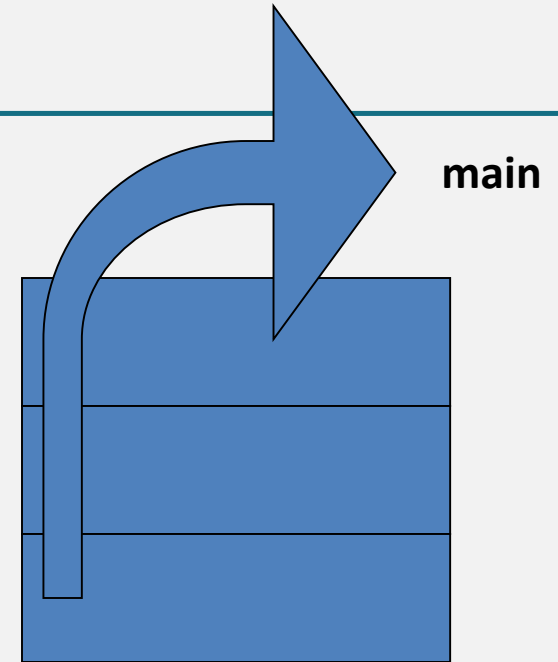
Follow from last
invocation.
There is nothing left

CALL STACK

Main Program

```
a = 2
int b = 5
printAbsdiv(a,b)
```

```
def printAbsdiv (a, b):
    res = 0
    res = a / abs(b)
    print(res)
```



End and back to
OS

CALL STACK

- All methods that are in the stack are said to be active
 - ✓ Execution is not finished yet.
- At any given time, all the active methods in an execution stack comprise what is called the execution stack trace (Stack Trace) of a thread.
- A program ends when...
 - ✓ There are not elements in the Call Stack
- Now let's get back to exceptions...

EXCEPTION MANAGEMENT

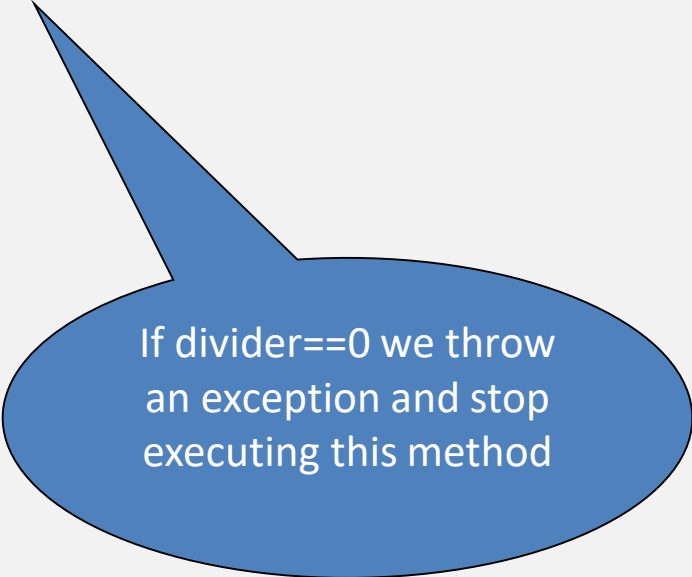
- With exceptions we can interrupt the execution flow of a method by throwing an exception until some of the methods that were invoked know how to handle the exceptional case
 - ✓ Alternative flow
- Let's see some reserved words

THROWING EXCEPTION

- To throw an exception inside a method we need one thing:
 - ✓ Throw the exception (if conditions are met) using the reserved word **raise**

THROWING EXCEPTION

```
def divide (dividend, divider):  
    res = 0;  
    if (divider == 0):  
        raise ZeroDivisionError      # or ArithmeticError  
    else  
        res = dividend / divider  
    return res
```



If divider==0 we throw an exception and stop executing this method

EXCEPTIONS – CLASSES AND OBJECTS

- There are different types of exceptions and each of them materializes as a class.
- When an exception is actually raised, an object of that class is instantiated or used.
- An exception **is an object** that reports a failure to perform certain operations during the execution of a program. Exceptions inherit from class “Exception”

THE OTHER SIDE OF RAISING

- Raising exceptions
 - ✓ We know the execution is interrupted
 - ✓ Where does the execution resume?
- Catching exceptions
 - ✓ It resumes in some method that can capture the exception, offering an alternative flow.
 - ✓ For this, we will see the clauses **try** – **except**

CATCHING EXCEPTIONS

- To catch an exception we need two things:
 - ✓ Block where statements can throw an exception
 - ✓ Using **try**
 - ✓ Block with statements to handle exceptions thrown by the “try block”.
 - ✓ Using **except**

CATCHING EXCEPTIONS

try:

 You do your operations here;

except ExceptionI:

 If there is ExceptionI, then execute this block.

except ExceptionII:

 If there is ExceptionII, then execute this block.

else:

 If there is no exception then execute this block.

- A single try statement can have multiple except statements.
- You can also a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection .

EXECUTION FLOW

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can\'t find file or read data" # block B
else:
    print "Written content in the file successfully"
    fh.close() # block C
```

- Within the try, statements are executed normally.
- If there are no exceptions thrown, the code will continue in C.
- If an exception of the type in the catch is generated, skip to execute the code in B.
- If an exception of another type is generated, the execution of this method is aborted, and the same exception is raised to the invoking method.

EXCEPTION AS A WAY TO DELEGATE

- The method that throws the exception delegates it to another place in the program
 - ✓ To throw: **raise**
- The method that catches the exception is the delegate in charge of handling it.
 - ✓ To catch: **try** – **except**

EXCEPT WITH NO EXCEPTIONS

```
try:
    You do your operations here;
    .....
except:
    If there is any exception, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

- This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

EXCEPT WITH MULTIPLE EXCEPTIONS

```
try:
    You do your operations here;
    .....
except(Exception1[, Exception2[,...ExceptionN]]):
    If there is any exception from the given exception
    list,
    then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

- This kind of a try-except statement catches all the listed exceptions

TRY-FINALLY CLAUSE

```
try:
    You do your operations here;
    .....
    Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

- The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.
- You cannot use else clause as well along with a finally clause.

3 BIG FAMILIES

- There are 3 kinds of exceptions
 - ✓ Errors
 - ✓ Checked Exceptions
 - ✓ Runtime Exceptions

ERRORS

- ***Errors***
 - ✓ Exceptional external conditions.
 - ✓ Out of memory
 - ✓ Hardware failure
 - ✓ Call Stack out of memory
 - ✓ **It is not necessary to wrap sentences that produce this type of error in try-catch blocks.**

CHECKED EXCEPTIONS

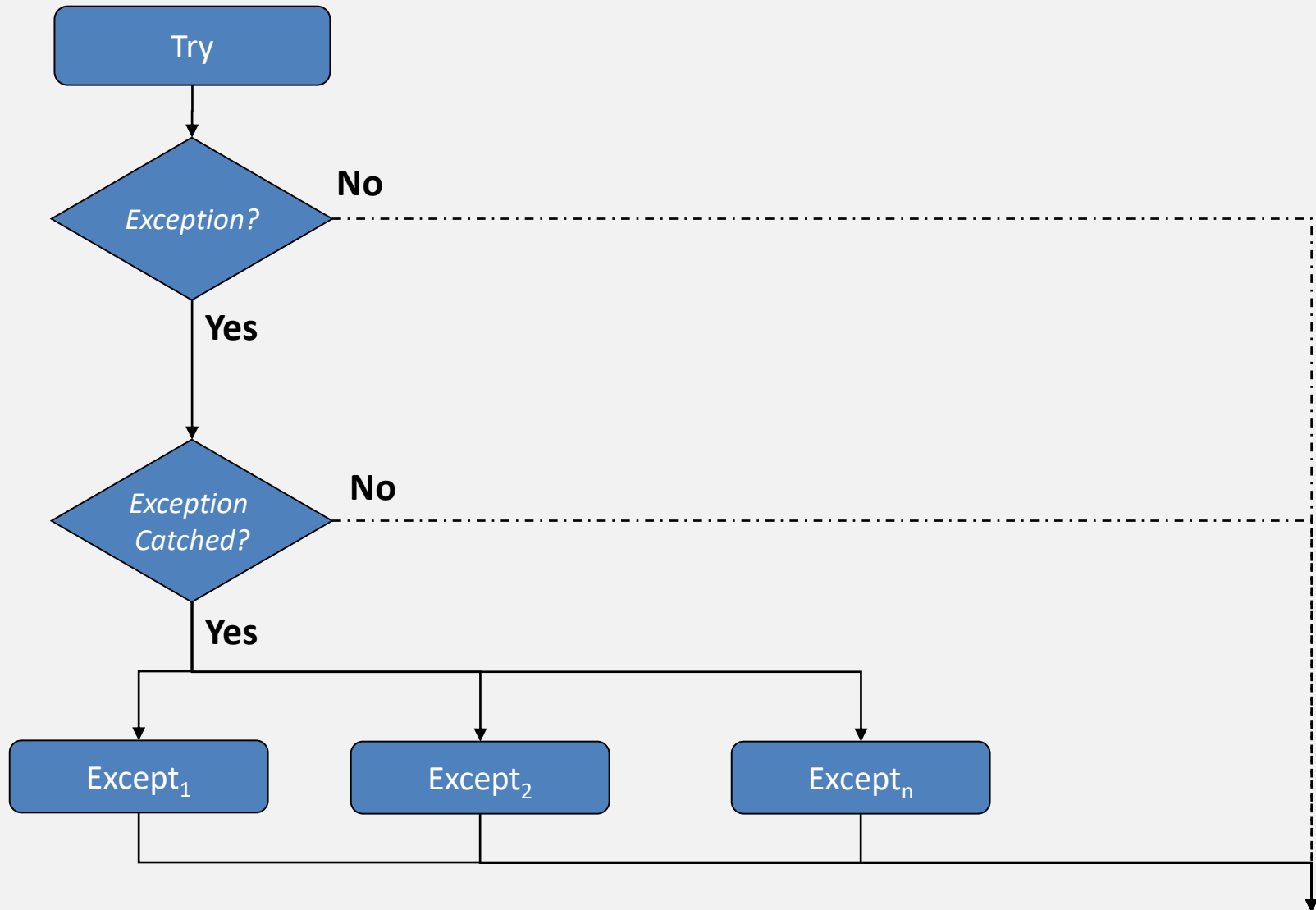
- *Checked exceptions*
 - ✓ Exceptional conditions that a well-written application must anticipate and deal with in case of.
 - ✓ File not found
 - ✓ Statements that produce these types of exceptions **must** be wrapped in try-except blocks.

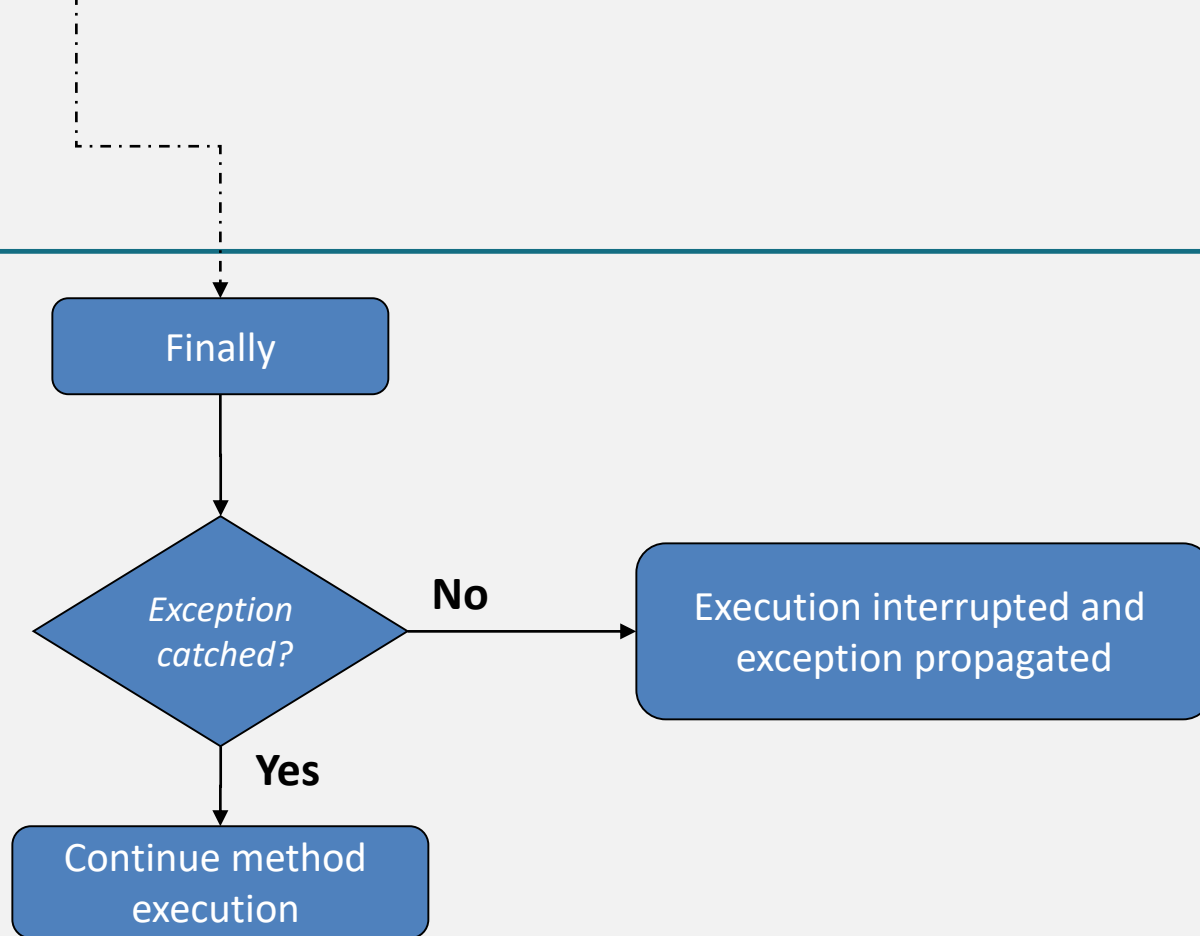
RUNTIME EXCEPTIONS

- ***Runtime exceptions***

- ✓ Exceptional conditions that an application can not anticipate, mostly due to programming errors.
 - ✓ `ArithmeticError`
 - ✓ `ZeroDivisionError`
 - ✓ `StopIteration`
 - ✓ `IndexError`
- ✓ It is not necessary to wrap sentences that produce such errors in blocks **try-except**.
- ✓ We don't need to declare them (**raise**)

COMPLETE FLOW





FINAL CONCLUSION

- The end user does not care which approach we use, but that the application works correctly.
 - ✓ Being robust means it works well even in hostile situations.

API'S

API'S

APPLICATION PROGRAM INTERFACE

- Set of classes, enums and other elements ready to be used in a program that is being developed.
 - ✓ They provide me with functionality that I need, avoiding to start from scratch
- There are different types of APIs depending on the size they have and features they provide

TOOLKITS

- A toolkit is a set of reusable classes designed to provide useful general purpose functionality.
 - ✓ Toolkits doesn't impose a specific design to our application
 - ✓ They only provide functionality
 - ✓ We avoid to develop things that are already developed.
 - ✓ Toolkits focus in code reutilization.

FRAMEWORKS

- A Framework is a set of cooperating classes that constitute a reusable base design for a specific category of software.
- When we use a framework we base our app on it
 - ✓ We start with the framework and we continue the development to create our application
 - ✓ Using some standards and formats from the framework.
- A framework determines the **architecture** of the application, the general structure, the partition in classes and objects, as they collaborate together with the control thread.

EXAMPLES

- Flexible Toolkit to build Web APIs
 - ✓ Django REST Framework
- Framework for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.
 - ✓ Kivy

THE END

QUESTIONS?

ComIT

OOP VII

THE TOPICS

- Abstraction
- Abstraction mechanisms
 - ✓ Classification
 - ✓ Association
 - ✓ Group
 - ✓ Composition
 - ✓ Aggregation
 - ✓ Dependency and Delegation
 - ✓ Generalization
 - ✓ Specialization

ABSTRACTION

- Abstraction is a thought activity where we have to focus on certain qualities of an entity ignoring others.
 - ✓ It is subjective and context dependent

ABSTRACTION

- To propose a computer-based solution to a real-life problem, we have to know very well what is the domain of the problem.

ABSTRACTION

- Abstraction mechanisms allow us to:
 - ✓ Classify
 - ✓ Rank
 - ✓ To categorize
 - ✓ Relate things in a conceptual way
- They are the first thing we have at hand to identify, analyze and design elements of my system based on existing relationships.

ABSTRACTION

- Abstraction serves to model the domain of a problem.
- We create a model of the current scenario considering that we are going run it on a computer.
- Modeling is a mixture of science, art and practice.
 - ✓ In OOP we focus on developing classes with which we construct objects that collaborate to achieve a certain goal.

ABSTRACTION MECHANISMS FROM AN OBJECT PERSPECTIVE

CLASSIFICATION

CLASSIFICATION (I)

- It consists of encapsulating objects in a given class.
- We group the objects by things they have in common and put them into conceptual categories
 - ✓ **Classes**
- Relates: **Object to Class.**

CLASSIFICATION (II)

- Example 1:
 - ✓ Furniture with legs and a surface parallel to the floor
 - ✓ **Table**
- Example 2:
 - ✓ Furniture with circular shape and with a cushion that supports a person
 - ✓ **Chair**
- Example 3:
 - ✓ Peter, my classmate
 - ✓ **Person.**
 - ✓ Could it belong to Student?
 - ✓ Yes. This process is subjective and context dependent

ASSOCIATION

ASSOCIATION (I)

- It consists of determining existing relationships that connect objects of different classes.
- We will see binary associations:
 - ✓ Links elements of 2 categories.
- Relates: **Objects to Objects.**

ASSOCIATION (II)

- Example 1:
 - ✓ One or more people live in one or more places.
- Example 2:
 - ✓ A person is married to another person.
- Example 3:
 - ✓ A **multiple choice** questions has several **options**.

ASSOCIATION (III)

- Example 4:
 - ✓ You have one, several or no courses in a classroom.
- Example 5:
 - ✓ One or more teachers teach one or several courses.
- An object of a class can be related in two different ways to an object in another.
- To create an association we must first classify the objects.
 - ✓ Define the object's classes

ASSOCIATIONS' PROPERTIES

- We need to identify for each object:
 - ✓ **Role**
 - ✓ **Cardinality**
 - ✓ With how many elements of the other category it is related
 - ✓ **Navigability**
 - ✓ Knows or uses related objects

TYPE OF ASSOCIATIONS

- When an association is modeled, sometimes we create classes that represent the relation between other classes.
 - ✓ Properties and functions from the relation.
 - ✓ And not from the objects being related
- Association is modeled as a class
 - ✓ Each object relates objects from other classes.

TYPE OF ASSOCIATIONS

Example 1:

- A student studies a discipline, each of them attends each discipline.
- **Course** relates **Student** with **Discipline**.
 - ✓ Partial grades
 - ✓ Final grade
 - ✓ Observations
 - ✓ Aptitude

TYPE OF ASSOCIATIONS

Example 2:

- We have a company in which people work.
- A job is something that associates the person with the work he does in the company
- **Employment** relates a Person to the employment contract he has in a Company.
 - ✓ Salary
 - ✓ Contract Date
 - ✓ Role

TYPE OF ASSOCIATIONS

Example 3:

- We have buses boarded by passengers.
- To travel, each passenger buys a ticket
- Ticket relates a bus with the trip that a passenger buys to travel in it.
 - ✓ Date
 - ✓ Amount
 - ✓ Start Stop and End Stop

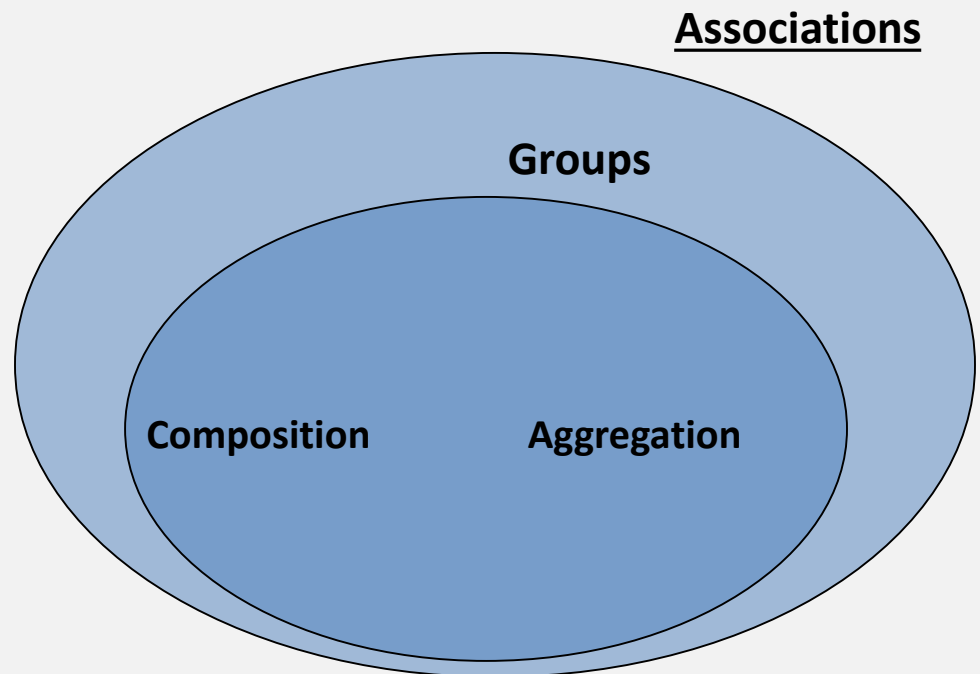
“SPECIAL” ASSOCIATION CASE

GROUP

- It is a relationship in which an object ("Container" or "all") groups another ("Contents" or "parts")
 - ✓ One of the objects is a preponderant factor in the relationship.
 - ✓ The "all" is at a conceptually higher level than the "part"
 - ✓ Relationships of type have, group, congregate,...

GROUP

- Group is a more specific concept than association
 - ✓ Subdivided according to the existential dependency
 - ✓ **Aggregation**
 - ✓ **Composition**



GROUP

- There is an existential dependency between the elements. If the Container element dies then so will Contents.

GROUP

- It is mandatory to speak of **cardinality** of the objects that intervene in the relationship.
 - ✓ **How many** objects are grouped by the container object

AGGREGATION

- Relationship that occurs when an object gathers another or others but is not the sole owner of it.
- There is no existential dependency between its elements.

AGGREGATION

Example 1:

- A screen, with keyboard, with Mouse, and the CPU, makes a computer. If the computer breaks, you can use the screen, the mouse, probably a large part of the CPU.

Example 2:

- Wheels, gears, windows, an engine, make a car. If a car is eliminated, maybe some auto parts can be used to build another car.

COMPOSITION

- Relationship that occurs when an object ("Compound") is formed by another or others ("Components") and is the sole owner of the same.
- There is an existential dependency between the elements.
- In the composition, the composite object should be solely responsible for managing the component objects.

COMPOSITION

Example 1:

- A checking account has operations. If I delete the current account then the operations would have no more reason to be.

Example 2:

- A "multiple choice" question contains several selectable options. If I delete the question these options will not make sense.

DIFFERENCES BETWEEN AGGREGATION & COMPOSITION

- A group, depending on the context, can be understood as composition or aggregation.
 - ✓ It is subjective and dependent on the context

IN SUMMARY

- Aggregation
 - ✓ Shared Contents
 - ✓ Existential independence
 - ✓ *(*) Relationship “Gathers”*
- Composition
 - ✓ Exclusive Contents (“composite”)
 - ✓ Existential dependency
 - ✓ *(*)Relationship “Composed of”*

BACK TO ASSOCIATION

- Mechanism to find relationships among objects.
 - ✓ Groups are associations with specific characteristics.

DEPENDENCY

- An association involves establishing a dependency between related objects.
 - ✓ If one is modified the other "will become aware" of it
- Types of dependency:
 - ✓ One depends on the other (Recommended)
 - ✓ Mutual or circular dependency

DELEGATION

- When an object depends on another one (s) contained, to fulfill a certain function, the dependency is known as **delegation**.

ASSOCIATION CLASSIFICATION

- Regardless of whether there is a group or not, binary associations can be defined:
- According to their **cardinality**:
 - ✓ One to one
 - ✓ One to Many or Many to One
 - ✓ Many to Many
- According to their **navigability**:
 - ✓ **Unidirectional**
 - ✓ **Bidirectionals**

MULTIPLE ASSOCIATIONS

- Are there associations in which more than two classes participate simultaneously?
 - ✓ Of course there are!
 - ✓ In practice it is usually expressed as several binary relations using association classes.

MULTIPLE ASSOCIATIONS

- A **player** is hired by a **team** for a **season**.
 - ✓ The **contract** establishes the relationship between 3 objects of different classes.
- It is mandatory to have objects that "tie" 3 objects of the other classes that in addition can have additional attributes.

SPECIALIZATION AND GENERALIZATION

SPECIALIZATION

- It is the mechanism through which you can subdivide a class into two or more classes, distinguishing particularities (things not shared) from the objects that belong to the class.
- Subcategories are created from a larger category.
- Mechanism relating: Class to Class.

SPECIALIZATION

- Specialization 1:
 - ✓ Furniture
 - ✓ Desk
 - ✓ Chair.
- Specialization 2:
 - ✓ Musical Instrument,
 - ✓ Guitar,
 - ✓ Flutes
 - ✓ Two pianos
 - ✓ Drum

SPECIALIZATION

- The key is to ask:
 - ✓ What are the differences between them?
 - ✓ Why are they part of the same hierarchy but not of the same class?
 - ✓ What details interest me?

GENERALIZATION

- Maybe, objects of two or more classes have many things in common so both belong, in essence, to the same class that synthesizes that which they share.
- A larger category is created from subcategories.
- Mechanism relating: Class to Class.

GENERALIZATION

- Generalization 1:
 - ✓ Desk
 - ✓ Chair
 - ✓ **Furniture.**
- Generalization 2:
 - ✓ Guitar
 - ✓ Drums
 - ✓ Bass
 - ✓ Violin
 - ✓ **Musical Instrument**

GENERALIZATION

- The key is to ask:
 - ✓ What do they have in common?
 - ✓ Can they be part of the same hierarchy?
 - ✓ What details do not interest me?
- I detach myself from the details and I abstract.

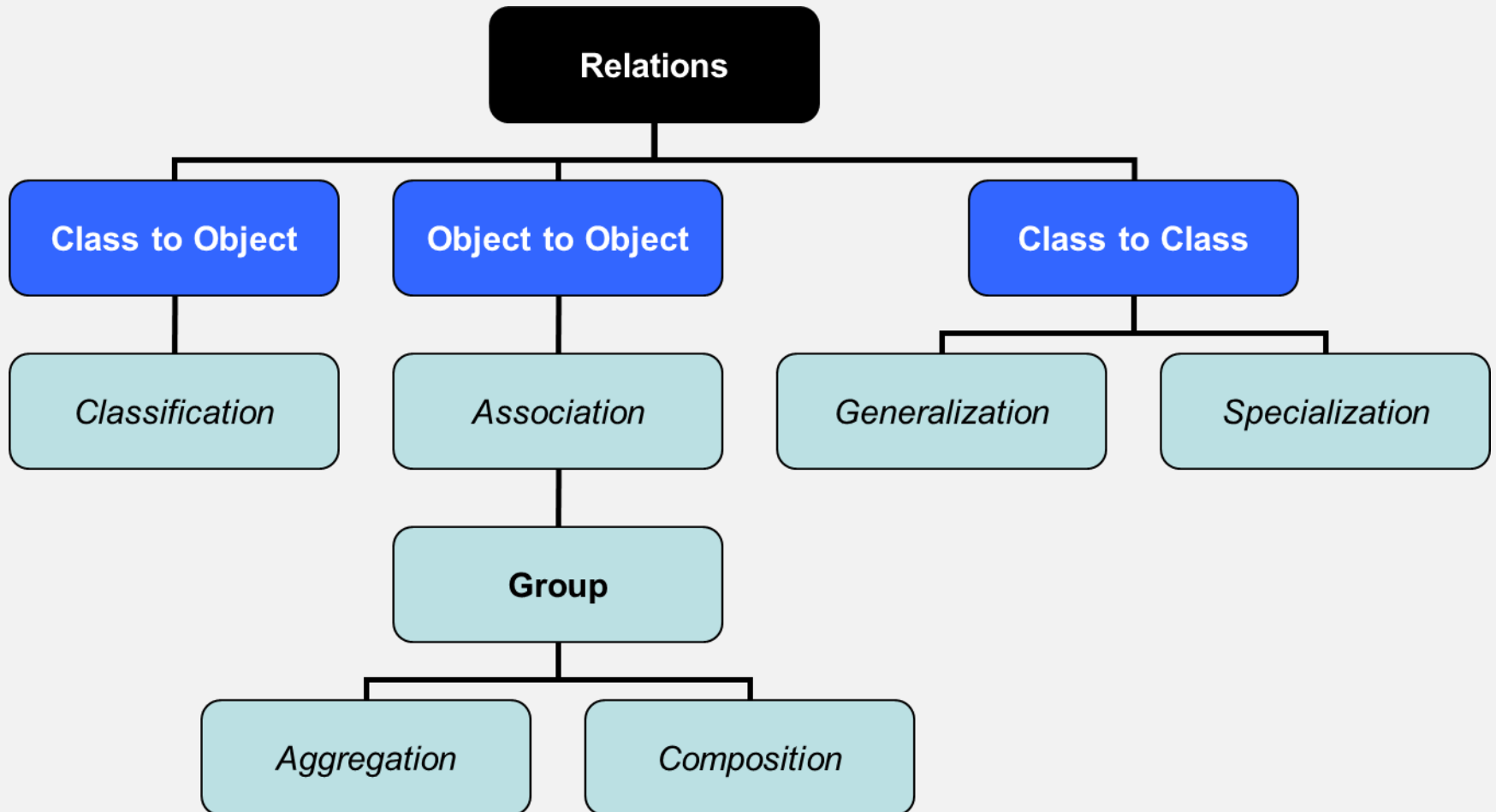
GENERALIZATION

- This mechanism establishes the famous rule of inheritance.
- If a class is a subclass of another, there is a relationship of the type "is a".

IN SUMMARY

- Object and Class
 - ✓ Classification
- Object and Object
 - ✓ Association
 - ✓ Group
 - ✓ Aggregation
 - ✓ Composition*
- Class and Class
 - ✓ Generalization
 - ✓ Specialization

HIERARCHICAL TABLE OF ABSTRACTION MECHANISMS IN OO



THE END

QUESTIONS?