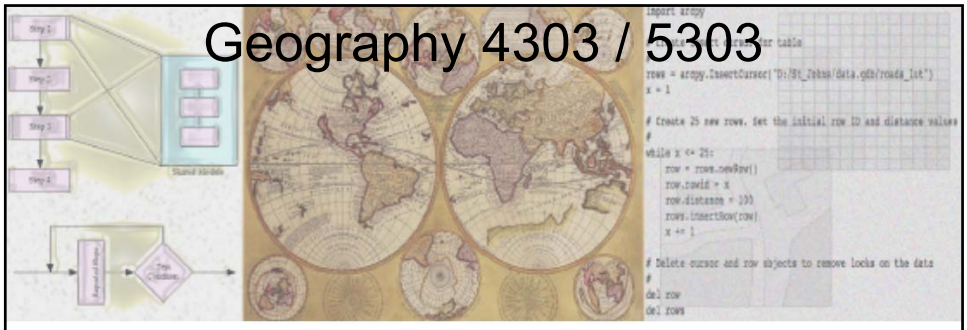


Geography 4303 / 5303

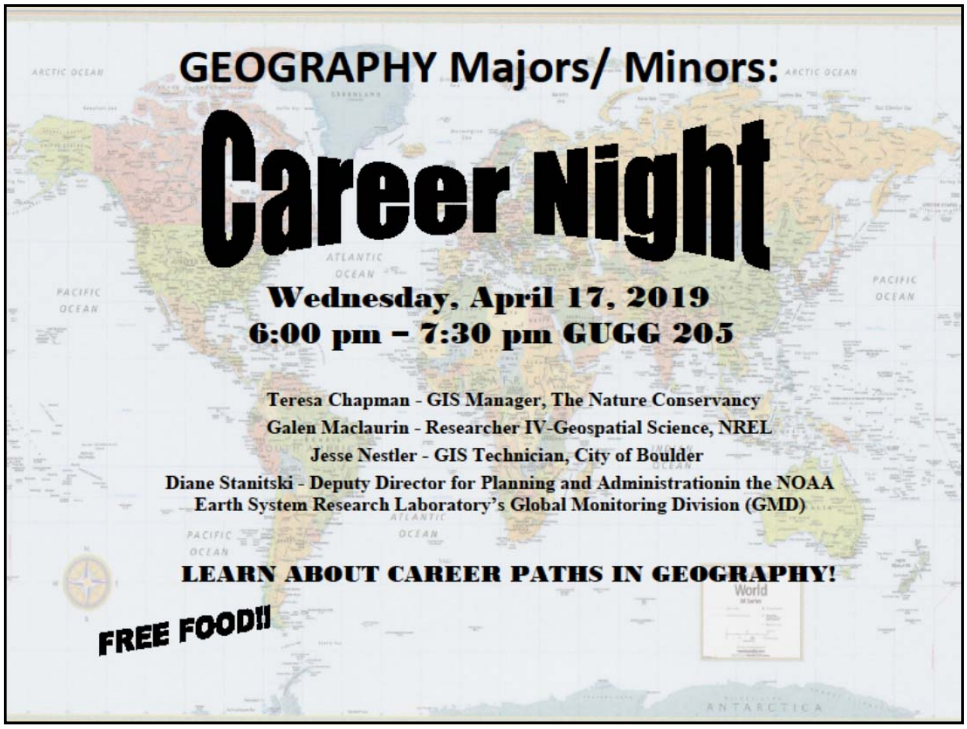
GIS PROGRAMMING FOR SPATIAL ANALYSIS

Class 10: Object Oriented Programming

Some Updates

- Final projects
- Stay in touch
- Project leaders' responsibilities
- Strategies and working packages
- Final paper

Lecture time in the next five weeks



Last Lecture / Last Week

- Arc raster analysis and objects
- Numpy
- Other modules/ tools
- Raster analysis could easily fill up its own course as you have seen - here we remain just scratch the surface…

Today ‘s Outline

- We will discuss **object-orientation** on a conceptual level
- You will learn what the **key elements** of object-oriented programming are
- You will understand why object-oriented programming can be useful and where the limitations for procedural programming can be found
- We will discuss a few **examples**

Learning Objectives

Answers to some questions like this

- What is an **object**?
- What is a **class**?
- What is an **instance**?
- What is **polymorphism**?
- What is **inheritance**?
- What is **encapsulation**?
- And finally: Why should we do that?

Part I
Object-oriented or not?

- What we know:
  - Python supports object-oriented technology
  - Python offers a full developing environment
- Python as object-oriented language?
- Geoprocessing and object-orientation
- Did you ever use objects while programming?

Characteristics of Object Oriented Programming

- **Objects** (instances of classes) as central elements
- **Data** and **functions** for communication and actions
- Code **reusability**
- **Program structure and generality**
- Principles that all OOP languages share:
  - ***-Abstraction***
  - ***-Encapsulation***
  - ***-Inheritance***
  - ***-Polymorphism***

What you already know …

- You worked already with classes/objects
- A **string object** stores the character sequence (the **data**) and provides ("built-in") **methods** to **operate** on the data (`s.find()`, `s.upper()`, `s.split()`)
- A **list object** stores the sequence of values/objects (the **data**) and provides ("built-in") **methods** to **operate** on the data (`l.extend()`, `l.append()`)

Classes & Objects

- A collection of **objects** of the same type (of similar characteristics)
- Represents an **abstraction** of the key elements of the system
- Objects are **instances** of classes ("a chicken is of type bird") – we "instantiate" an **object!**
- **References** to these objects are stored in variables

List! & Person code 01

Objects & encapsulated data and functions

A class can contain:

- **Data** (defining the **states**), and
- **Functions** (defining the **behavior**)
- Other classes
- Functions **operate** on that data ("methods")

- While working with an **object** you get access to its **data attributes** and **methods**, too
- These are "contained" or **encapsulated** in the object

Person code 01

States and behavior…

- **States** are properties; things that characterize something:
  "a **yellow** car", "a **hot** pan", "a **1.5m** tall person"
- **Behaviors** are things that something **does**:
  "a car **drives** ", "a bird **flies**", "a person **grows**"
- **Behaviors** can **change** properties:
  "The person **grows** and is **1.8m** tall."

States/behavior of Lists?

Person code 01

…through class methods

- **Behavior** associated with a class
  - Changing properties
  - Reporting values of properties (return values)
  - Performing actions in response to requests
- **Instances** of a class to call the method `MyInstance.myMethod()`

States/behavior of Lists?

Person code 01

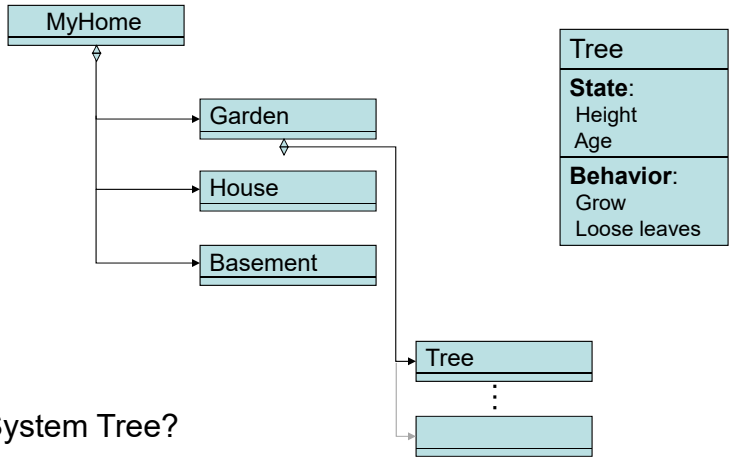
Tree: Abstracting a system

- "**abstract**" (key) **elements** encapsulated
- **States?**
- **Behavior?**
- Class as a **template** to define the "kind of" **behavior** of the system to be modeled

Show tree code

Modeling a System

Tree as part of Garden, Forest, Orchard,… www.uml.org



System Tree?

How to Work with Classes in Python

- **Defining user-specified** types
- Objects (**instances**) for access to **states** and **behavior**
- **Instantiation** is done via
  `MyObject = myModule.MyClass()`
- Using **methods**:
  `MyObject.myMethods()`

Show Tree code

Defining classes: `constructor`

Name of the class: Tree

Constructor: `__init__` is a special method that is called when a **new instance** of the class is created (constructed)

Variables inside `__init__` are **unique** to the new instance - and you can pass **parameters**

```
class Tree:
    def __init__ (self, val):
        self.val = val
        print "Class instance of Tree created! ID: ", self.val
>>> import class21ClassTree
>>> class21ClassTree.Tree(12)
Class instance of Tree created! ID:  12
```

Example Tree…

Defining classes: `self`

- Method definition: `self` parameter (`this` in J)
- **SELF: Reference to this one object instance** (filled in by the interpreter when you call the methods without it)
- Internal **reference** to **`itself`**
- …to pick up the **instance data**

`MyObject.doIt(arg1) => MyObject.doIt([[self]],arg1)`

```
class Tree:
    def __init__ (self, val):
        self.val = val
        print "Class instance of Tree created! ID: ", self.val
>>> import class21ClassTree
>>> class21ClassTree.Tree(12)
Class instance of Tree created! ID:  12
```

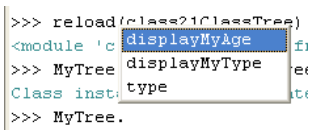
Defining classes: `Class methods`

```
class Tree:
    def __init__ (self, type):
        self.type = type
        print "Class instance of Tree created!"
    def displayMyType(self):
        print "I am a ",self.type
    def displayMyAge(self, age):
        print "I am ", age, " years old."
```

```
>>> reload(class21ClassTree)
<module 'class21ClassTree' from 'class21ClassTree.py'>
>>> MyTree = class21ClassTree.Tree("pine")
Class instance of Tree created!
>>> MyTree.displayMyType()
I am a  pine
>>> myArg = 365
>>> MyTree.displayMyAge(myArg)
I am  365  years old.
```

Functions that belong to a class (not independent)

Pars and Args



Class and object variables

- **Ownership** of variables
- **Class variable** can be accessed by **all** objects (instances) – “Shared”
- **Object variable**: **Each object** owns a copy of this variable (not shared between objects)

Class and object variables

```
class Tree:
    population = 0
    def __init__ (self, type):
        self.type = type
        print "Class instance of Tree created!"
        Tree.population += 1
    def displayMyType(self):
        print "I am a ",self.type
    def displayMyAge(self, age):
        print "I am ", age, " years old."
    def countPopulation(self):
        print "We are ", Tree.population, " trees."
```

```
>>> reload(class21ClassTree)
<module 'class21ClassTree' from 'class21ClassTree.py'>
>>> MyTree = class21ClassTree.Tree("pine")
Class instance of Tree created!
>>> MyTree.displayMyType()
I am a  pine
>>> MyTree.countPopulation()
We are  1  trees.
>>> MyTree = class21ClassTree.Tree("oak")
Class instance of Tree created!
>>> MyTree.displayMyType()
I am a  oak
>>> MyTree.countPopulation()
We are  2  trees.
```

- self.objectVar
- Tree.classVar

Summary Part I

- Some introductory thoughts of object-oriented concepts in programming
- This is the way to define your **own types** (**classes**), to create instances of these classes (**objects**) and assign them to variables
- Messages to the objects trigger **methods** we defined
- There is an important difference between **object** variables and **class** variables

Let's grow the tree… Area?

Part II
Two Important Features…

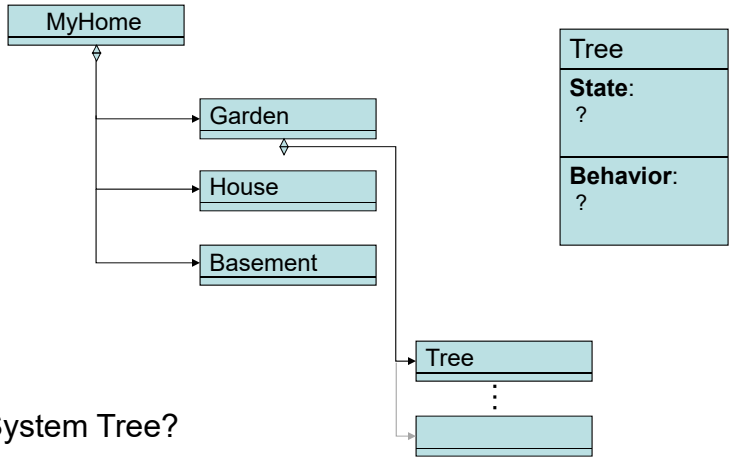
… of object-oriented programming:

- Polymorphism - "A single entity can take on multiple forms"
- Inheritance - "Use what has already been defined"

They represent key elements for object oriented programming

Modeling a System

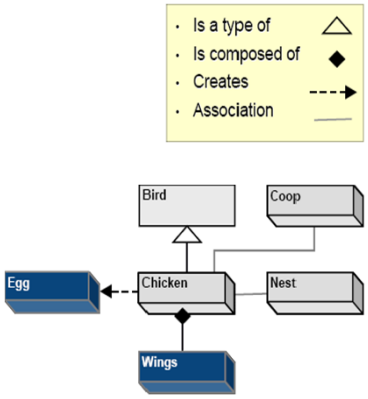
Tree as part of Garden, Forest, Orchard,… www.uml.org



System Tree?

Relationships between classes

- **Association**: „May have a relationship to...“
- **Type inheritance**: „Is a type of...“
  subclass - superclass; inheriting (plus own) props and meths
- **Composition**: „Is composed of...“
  „Whole-part“ relationship between classes
- **Aggregation**:
  „Whole-part“ relationship between classes
- **Instantiation**: „Creates...“
  An object of a class can be created (e.g., using a method of an object of another class)



Polymorphism

- **“Operators”** can work differently with objects of **different classes** (types)
- Two objects support the same **set of messages** (the same method calls) but their **methods** (of the same name and the same par set) do different things
- These objects can be **treated the same way** and yet will behave **differently**

Polymorphism

Overloading the concatenation operator "+":
Different functions depending on the implementation

Integer Addition:	1 + 2 = 3
Floating Point Addition:	3.14 + 0.0015 = 3.1415
List Concatenation:	[1, 2] + [4, 5, 6] = [1, 2, 4, 5, 6]
String Concatenation:	"foo" + "bar" = "foobar"

Polymorphism

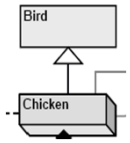
```
class Cat:
    def __init__(self, name):
        self.name = name
    def makeNoise(self):
        return 'Miau!'

class Dog:
    def __init__(self, name):
        self.name = name
    def makeNoise(self):
        return 'Wau'

a = Cat('Catty')
b = Cat('Trixi')
c = Dog('Trevor')

for myPet in [a, b, c]:
    print myPet.name + ': ' + myPet.makeNoise()
```

```
Catty: Miau!
Trixi: Miau!
Trevor: Wau
```

- Objects of **different types** respond to method calls of the **same name** showing **type-specific behavior**

Inheritance

- Classes (**subclasses**) can inherit **properties** and **methods** from the class they are derived from (**super or parent class**)
- "**type-subtype**" relationship
- Often used to implement **polymorphism**
- Reduced **re-implementation effort** if classes are similar
- **Inheriting** the capabilities needed and **override** those that are supposed to do different things




```
class Pet:
    def __init__(self, name):
        self.name = name
    def move(self):
        print "I can run and jump."

class Cat(Pet):
    def makeNoise(self):
        return 'Miau!'
    def move(self):
        Pet.move(self)
        print "I can climb."

class Dog(Pet):
    def makeNoise(self):
        return 'Wau'
    def move(self):
        Pet.move(self)

a = Cat('Catty')
b = Cat('Trixi')
c = Dog('Trevor')

for myPet in [a, b, c]:
    print myPet.name + ': ' + myPet.makeNoise()
    myPet.move()
```

- A parent class Pet has a method prototype move()
- Objects of type Cat/Dog instantiated inherit name and move()

```
Catty: Miau!
I can run and jump.
I can climb.
Trixi: Miau!
I can run and jump.
I can climb.
Trevor: Wau
I can run and jump.
```

Overriding Polymorphism

```
class Pet:
    #moveType = "any kind"
    def __init__(self, name):
        self.name = name
        print "instance of pet"
    def move(self, moveType):
        print "I can:", moveType

class Cat(Pet):
    def move(self, moveType):
        Pet.move(self, moveType)

class Bird(Pet):
    def move(self, moveType):
        Pet.move(self, moveType)

myCat = Cat('Catty')
myBird = Bird('Trevor')

print myCat.name, ': '
myCat.move("jump and run and climb")
print myBird.name, ': '
myBird.move("fly")
```

- Cat and Bird both **inherit** move() from Pet, but their derived class methods **override** the method of the **parent** class
- Manipulation vs. Extending

```
Catty :
I can: jump and run and climb
Trevor :
I can: fly
```

Multiple Inheritance

- If more than one class is listed in the **inheritance list**
- Make use of capabilities of more than one **parent class**
- E.g. Dog inherits from a Pet parent class as well as from a Mammal parent class

What are the Points…?

- **Polymorphism** allows client programs to be written based only on abstract interfaces
- Objects will make use of these interfaces (interface **inheritance**)
- Creating new types of objects, if the new objects conform to the original interface

Summary Part II

- **Inheritance** is all about using **existing** capabilities in an efficient way
- **Reusing** constructs already developed for similar cases (Do not reinvent the wheel)
- **New** objects (types) instantiated can make use of the functionalities already provided and **manipulate** them
- Decreasing complexity in programming (writing)
- Shared control code

Time for an exercise…