Programming I

Course 9

Introduction to programming

What we talked about?

- Modules
- List Comprehension
- Generators
- Recursive Functions
- Files

What we talk today?

- Object Oriented Programming
- Classes
- Objects

Object Oriented Programming

- Python is an object oriented programming language

- Way?
  - Fits a object oriented programming language definition

- A language or a technique is object oriented if and only if it directly supports [Stroustrup, 1995]
    - Abstractization – providing some form of classes and objects
    - Inheritance - providing the ability to build new abstractions out of existing ones
    - Runtime polymorphism – provide some form of runtime binding

Object Oriented Programming

- Terminology
  - Abstractization
    - Possibility to add user defined data types (new abstractizations)
  - Inheritance
    - providing the ability to build new abstractions out of existing ones
  - Polymorphism
    - Process objects differently based on their data type
  - Classes
    - Describe one or more objects
    - A template for creating, or instantiating, specific objects within a program.
  - Objects
    - A realization of the class

Objects

- Example on python objects
  - "Hello" <- object of type a string
  - [1, 2, 3, 4] <- object of type list
  - {"Programming", "Course" } <- object of type set
- Each object is characterized by
  - A unique identifier
  - A type
  - A internal representation
  - A set of operations that allows interaction with the information stored in the object

What can you do with objects?

- Create new objects
- Manipulate objects
- Destroy objects
  - explicitly using `del` or just “forget” about them
  - python system will reclaim destroyed or inaccessible objects – called “garbage collection”

What are objects?

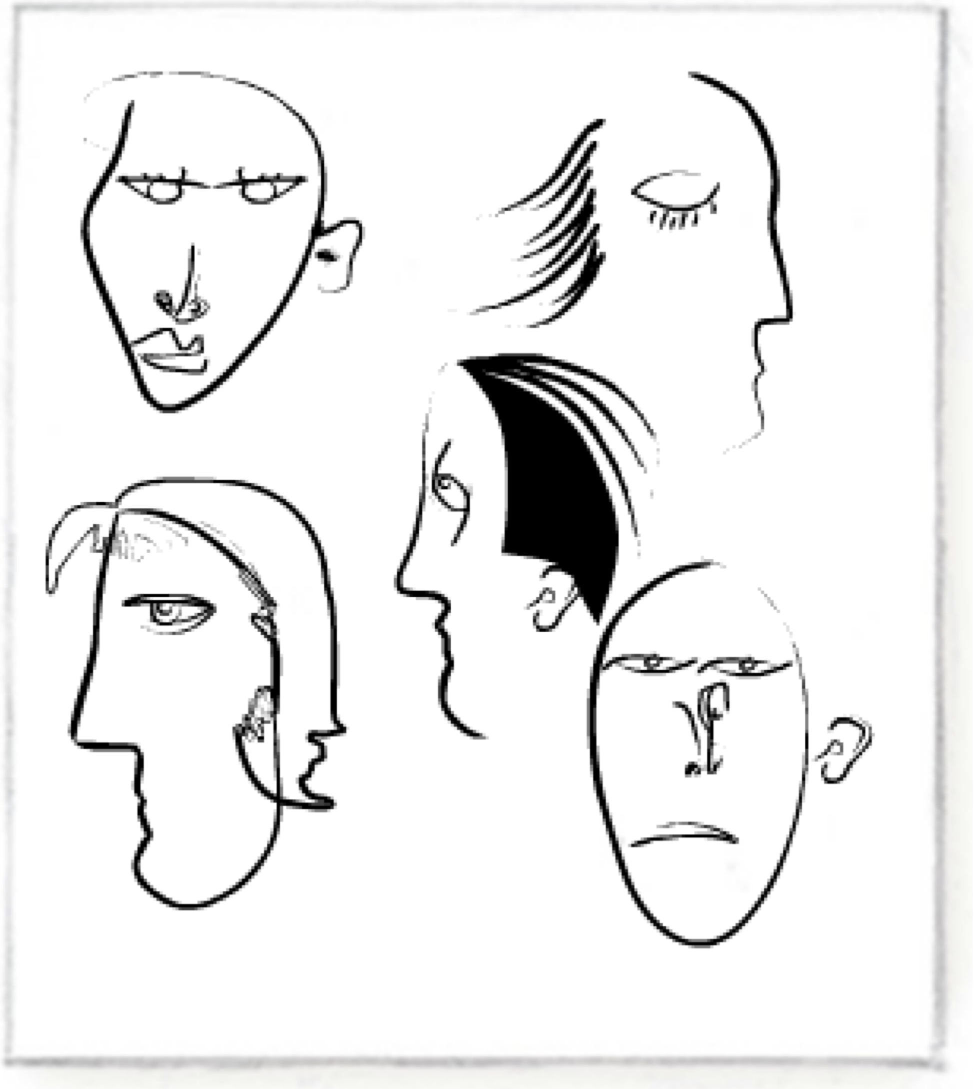
- A realization of an abstract concept that incorporates
  - An internal representation
    - Through data attributes values
  - An interface for interacting with objects
    - Through methods (aka functions or procedures)
    - Define behavior but hides implementation details
- How can be an object?
  - UVT University
  - The bank transaction that deposed 100 RON from mother account to child account

Example

- Python lists
  - How they are internally represented?
    - Dynamic array => object attributes
  - How you can manipulate them?
    - Object methods
    - `L[i], L[i:j], +`
    - `len(), min(), max(), del(L[i]),`
    - `L.append(),L.extend(),L.count(),L.index(),L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`

Example

- Humans heads
    - How many objects we have?
    - Could we provide a description that fits all heads?
    - There is a type in python that proper describe this type of objects?
- Image generated with http://www.picassohead.com/create.html



Classes – Own Types

- Describe similar objects
- Define classes that involves
  - Defining class name
  - Defining class members
    - Attributes
    - Methods
- Use classes that involves
  - Creating new instances (objects)
  - Applying operations on objects

Make a distinction between **creating a class** and **using an instance** of the class

Define Your Own Type

- Use the `class` keyword to define a new type

Class name
New data type name

Parent class name
Can be omitted and is considered by default object

```
class Coordinate (object):
     #define class members here
```

Class definition

- Similar to `def`, indent code to indicate which statements are part of the class definition

- The word `object` means that *Coordinate* is a Python object and inherits all its attributes (inheritance next lecture)
    - *Coordinate* is a subclass of object
    - `object` is a superclass of *Coordinate*

What are class members?

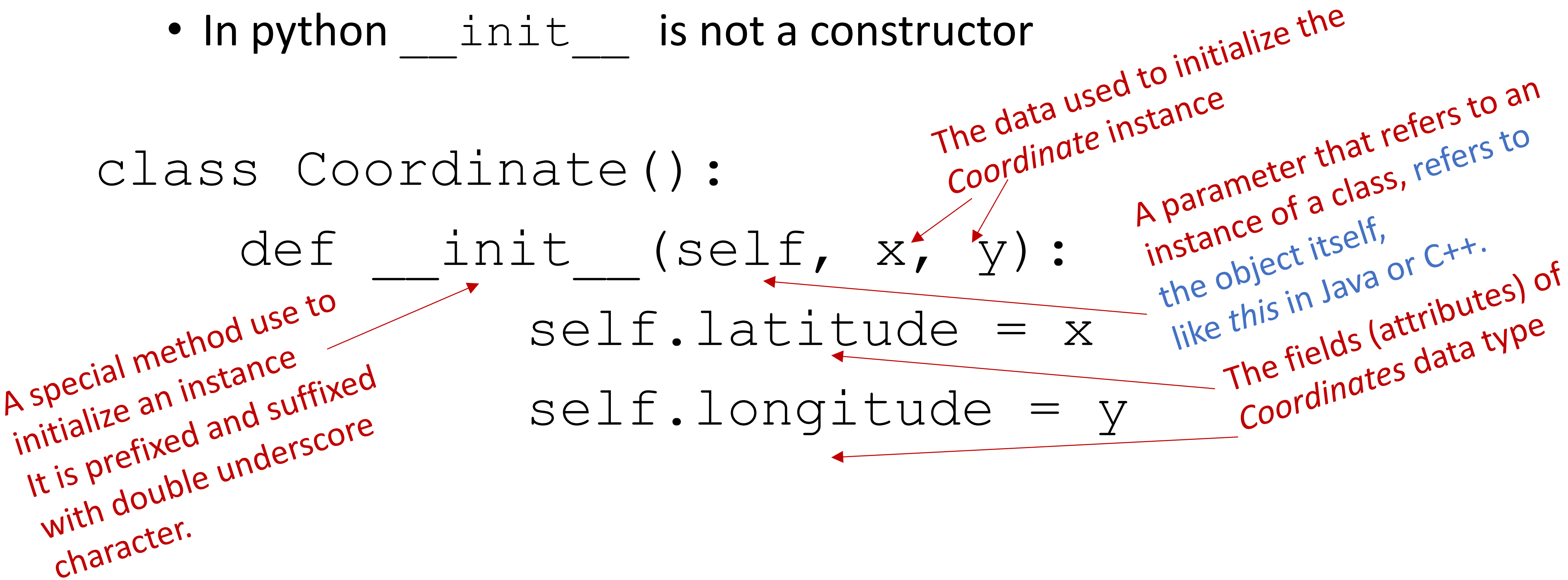
- Data and methods that "belong" to the class

- Data members (aka class fields, class attributes)
  - Data (variables) that describe the class
  - In case of *Coordinate* could be the latitude and longitude of a point on the globe

- Methods (aka member functions)
  - Allow to manipulate the data stored in the class
  - Allow interaction with other objects
  - Example
    - Display the coordinate like a real value or in degree, minutes and seconds
    - Calculate the distance between two coordinates

Defining How to Create an Instance of the Class

- Before using the new class we have to define how to create an instance of the object
- Use a special method called `__init__` to initialize class fields
  - In python `__init__` is not a constructor



```
class Coordinate():
    def __init__(self, x, y):
        self.latitude = x
        self.longitude = y
```

The data used to initialize the *Coordinate* instance

A parameter that refers to an instance of a class, refers to the object itself, like *this* in Java or C++.

The fields (attributes) of *Coordinates* data type

A special method use to initialize an instance It is prefixed and suffixed with double underscore character.

Define a Method for the *Coordinate* Class

```
class Coordinate():
    def __init__(self, x, y):
        self.latitude = x
        self.longitude = y
    def distance(self, other):
        x_diff = self.latitude - other.latitude
        y_diff = self.longitude - other.longitude
        return (x_diff**2+y_diff**2)**0.5
```

Used to refer to any instance

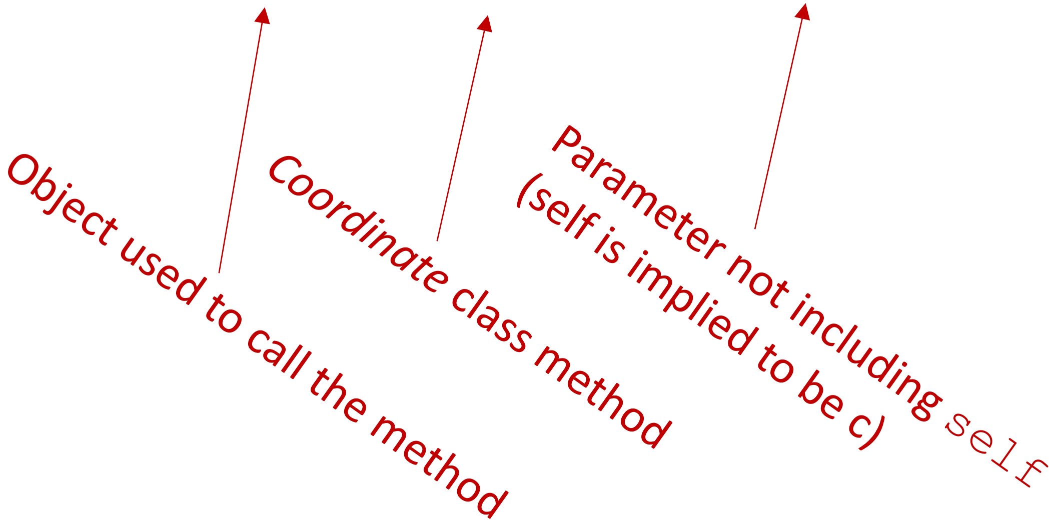
Another method parameter

Dot notation to access data

How to Use the Method

Conventional Way (used by most OO languages)

```
c = Coordinate(45, 45)
zero = Coordinate(0, 0)
print(c.distance(zero))
```



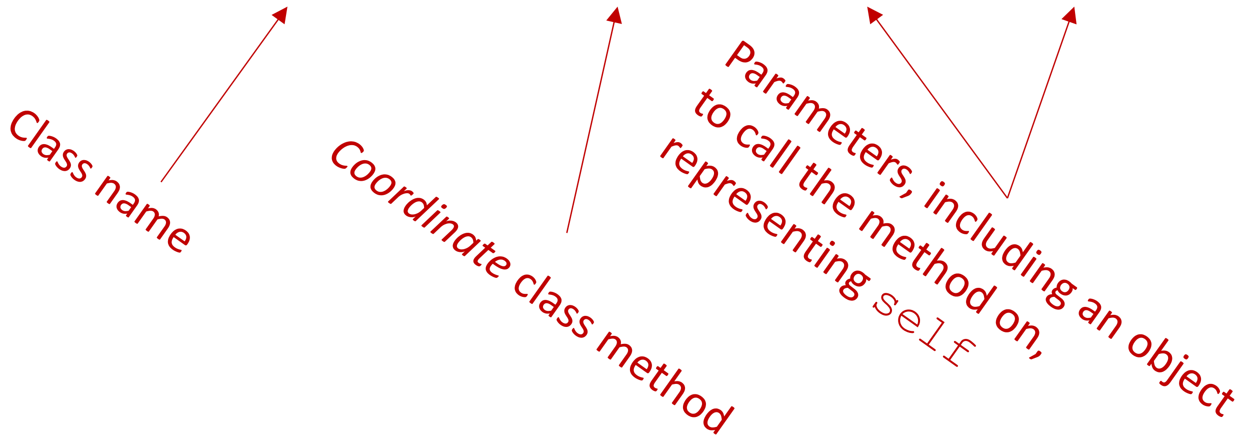
Object used to call the method

Coordinate class method

Parameter not including `self` (self is implied to be c)

Equivalent to

```
c= Coordinate(45, 45)
zero = Coordinate(0, 0)
print(
  Coordinate.distance(c, zero))
```



Class name

Coordinate class method

Parameters, including an object to call the method on, representing `self`

Printing Objects

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- Uninformative `print` representation by default
- Define a `__str__` method for a class
- Python calls the `__str__` method when used with print on your class object
    - Describe the way in which you want to see the details about an object

```
>>> print(c)
<3,4>
```

Printing Objects

```
class Coordinate():
    def __init__(self, x, y):
        self.latitude = x
        self.longitude = y
    def distance(self, other):
        x_diff = self.latitude - other.latitude
        y_diff = self.longitude - other.longitude
        return (x_diff**2+y_diff**2)**0.5
    def __str__(self):
        return "<"+self.latitude+","+self.longitude+">"
```

Special method name

Printing Objects

Class user use str to convert an object to string

Developers implement repr in order to offer a string representation for class objects

```
>>> c1 = Coordinate(3,4)
>>> c2 = Coordinate(3,4)
>>> l = [c1, c2]
>>> print(l)
[<__main__.Coordinate object at 0x10ebb1fd0>,
<__main__.Coordinate object at 0x10ebbc0f0>]
```

- `object.__repr__(self)`: called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object.
- `object.__str__(self)`: called by the `str()` build-in function and by the print statement to compute the "informal" string representation of an object.

Finding Information About Class Objects

- Can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3, 4>
>>> print(type(c))
<class __main__.Coordinate>
```

Result of `__str__` *method call*

The type of object c is a class Coordinate

- This makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

A Coordinate is a class

A Coordinate class is a type of object

- Use isinstance() to check if an object is a Coordinate

```
>>> print(isinstance(c, Coordinate))
True
```

Special Operators

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- Like print, can override these to work with your class
- Define them with double underscores before/after
  - `__add__(self, other)`
  - `__sub__(self, other)`
  - `__eq__(self, other)`
  - `__lt__(self, other)`
  - `__len__(self)`
  - `__str__(self )`
- ... and others

Special Operators

- Operator overloading
  - Allow classes to define their own behavior with respect to language operators

- Python approach to operator overloading
  - Implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names.

Special Operators - Example

- Create a new type to represent a number as a fraction
- Internal representation is two integers
  - Numerator
  - Denominator
- Interface a.k.a. methods a.k.a how to interact with `Fraction` objects
  - add, subtract
  - print representation
  - convert to a float

Public and Private Data

- All attributes of Coordinate class are public so it it possible to set them with undesirable values

  ```
  >>> c = Coordinate(3,4)
  >>> c.latitude = "a string"
  >>> print(c)
  <'a string', 4>
  ```

  Break distance function

- We therefore need to protect the `c.latitude` and provide accessors to this data
  - Encapsulation or Data Hiding
  - Accessors are "gettors" and "settors"
- Encapsulation is particularly important when other people use your class

Public and Private Data

- In Python anything with two leading underscores is "private"
    - __a, __my_variable
    - Still can be access by a Python trik
        - Coordinate
- Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.
    - _b
    - Sometimes useful as an intermediate step to making data private

=>

INFORMATION HIDING – making class attributes not accessible directly by user in order to not set them with undesirable values

GET/SET Methods

- Get "type" methods return the value of class attribute
- Set "type" methods put value in a class attribute

```
class Coordinate():
    def __init__(self, x, y):
        self.set_latitude(x)
        self.__longitude = y
    def get_latitude(self):
        return self.__latitude
    def set_latitude(self, x):
        if x<-90 or x>90:
            raise ValueError "Latitude values not valid"
        self.__latitude = x
```

User can obtain *latitude* value

Assure that only approved values can be set to *latitude*

User can set *latitude* value

Private attribute

GET/SET Methods

- Get "type" methods return the value of class attribute
- Set "type" methods put value in a class attribute

```
class Coordinate():
    def __init__(self, x, y):
        self.set_latitude(x)
        self.__longitude = y
    def get_latitude(self):
        return self.__latitude
    def set_latitude(self, x):
        if x<-90 or x>90:
            raise ValueError "Latitude values not valid"
        self.__latitude = x
    latitudine = property(get_latitudine, set_latitudine)
c = Coordonate(3, 4)
c.latitudine
```

Python feature that will redirect all variable modifications action trough set/get methods

Encapsulation

- One of the big benefits of classes is that they hide implementation details from the user => encapsulation.

- A well designed class has methods that allow the user to get out all the information they need out of it.
  - This allows a user to concentrate on their code rather than on your code.

- This also frees you to change the internal implementation of the class
  - Write to the Interface, not the the Implementation
  - Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public function

Encapsulation

- To encode related data, routines and definitions in a class capsule

- The interface is the visible surface of the capsule
  - The interface describes the essential characteristics of objects of the class which are visible to the exterior world

- The implementation is hidden in the capsule
  - The implementation hiding means that data can only be manipulated, that is updated, within the class, but it does not mean hiding interface data

Class Conventions

- Class names start with upper case letters.
  - In most cases are nouns at singular number

- Class methods and instances start with lower case letters.

- Method definitions should have docstrings just like function definitions.

- Classes should have docstrings just like modules have docstrings that describe what the class does.

Advantages of OOP

- Bundle data into packages together with procedures that work on them through well-defined interfaces

- Divide-and-conquer development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity

- Classes make it easy to reuse code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

Bibliography

- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

- http://www.cs.toronto.edu/~quellan/courses/summer11/csc108/lectures.shtml