# OOP: An Example

Using Inheritance

# USING **INHERITANCE**

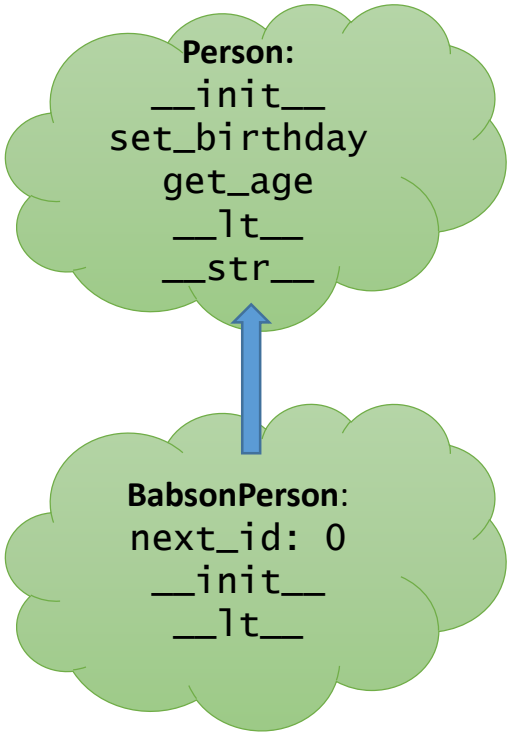
- Explore in some detail an example of building an application that organizes info about people

- Start with a `Person` object
  - `Person`: name, birthday
  - sort by last name
  - get age

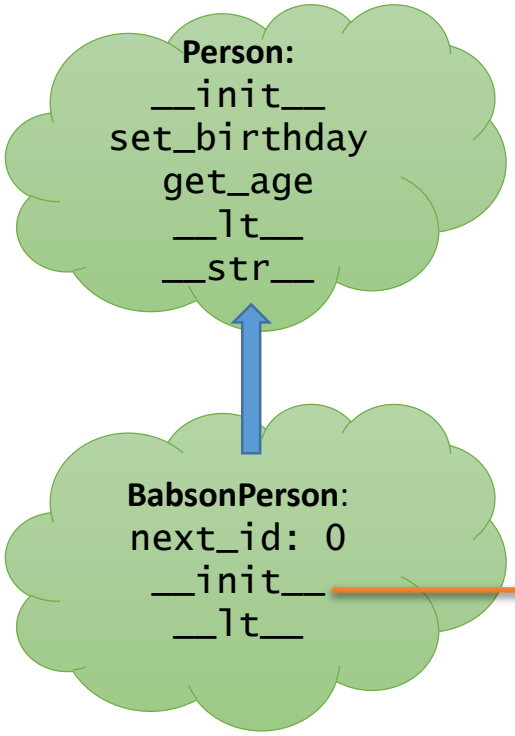
# USING INHERITANCE

- Explore in some detail an example of building an application that organizes info about people
  - Person: name, birthday
  - sort by last name
  - get age
- BabsonPerson: Person + ID Number
  - assign ID numbers in sequence
  - get ID number
  - sort by ID number
- See code in BabsonPerson.py

# VISUALIZING THE HIERARCHY

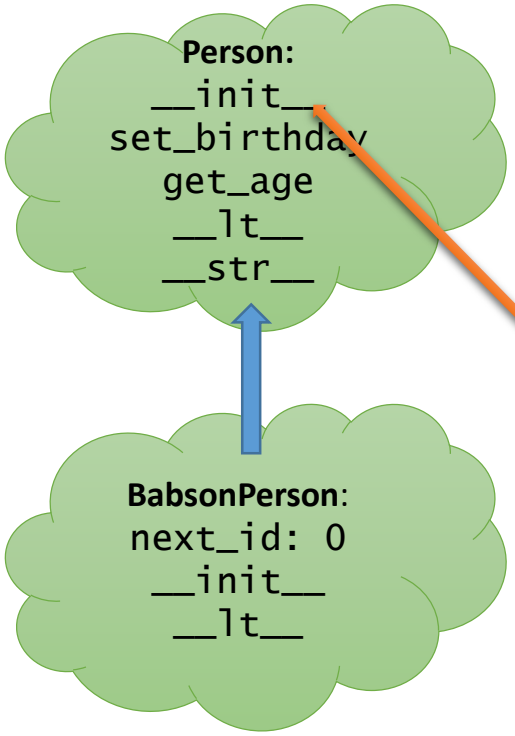


# VISUALIZING THE HIERARCHY



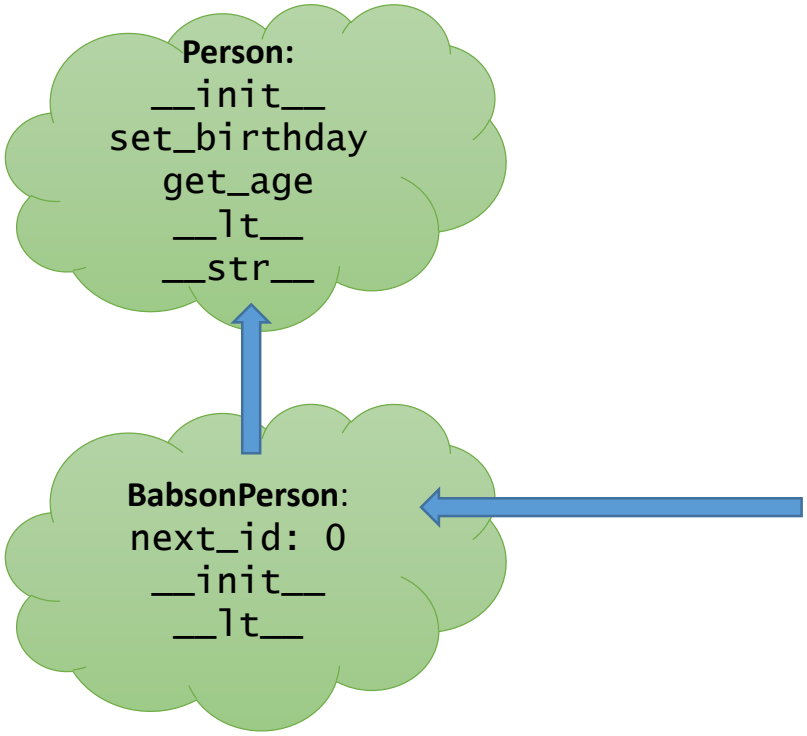

Calling `BabsonPerson` will use this `__init__` procedure (because that is the one visible in `BabsonPerson`'s environment)

# VISUALIZING THE HIERARCHY




That code uses `Person.__init__` which will in turn call this procedure.

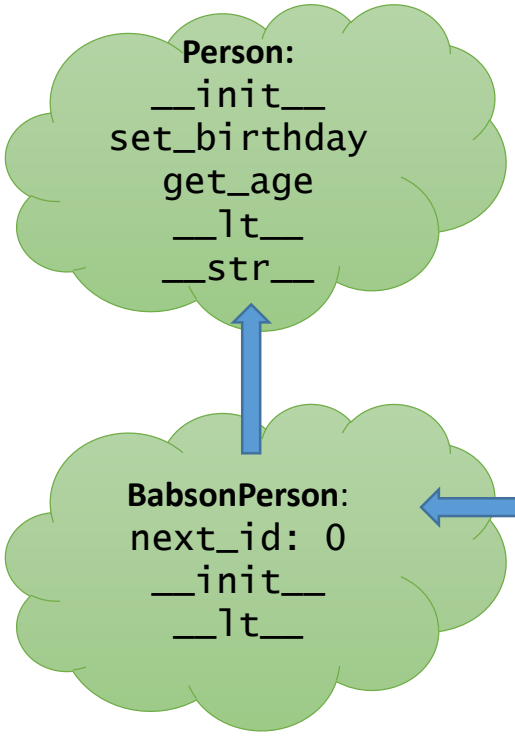
# VISUALIZING THE HIERARCHY




And that creates an instance of `BabsonPerson` (because of the first call, which inherits from the class definition) but with bindings set by the inherited `__init__` code

| Attribute | |
|---|---|
| birthday | |
| last_name | |

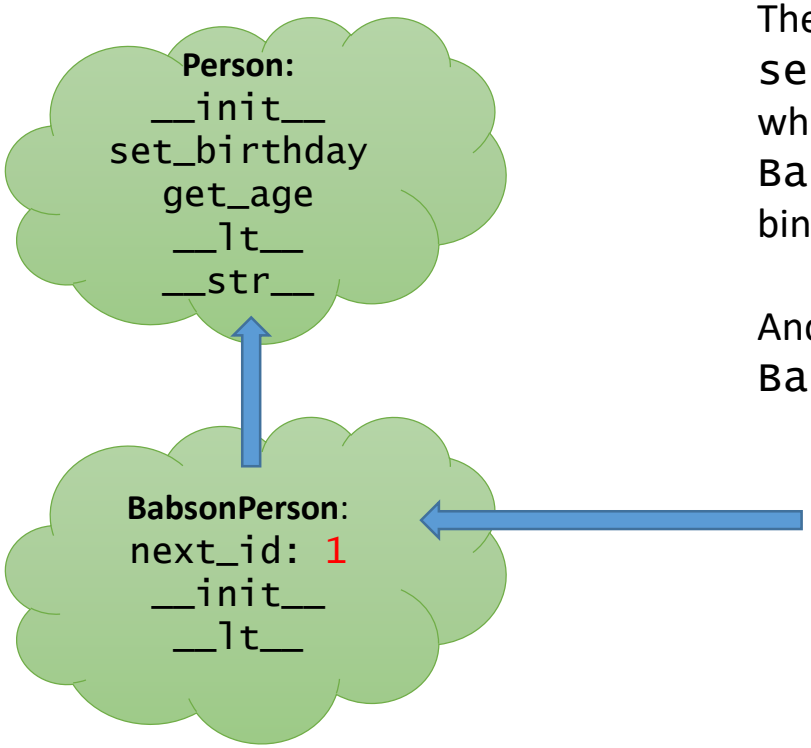
# VISUALIZING THE HIERARCHY




The rest of the original `__init__` code calls `self.id= BabsonPerson.next_id` which looks up `next_id` in the `BabsonPerson` environment, and creates a binding in `self` (i.e. the instance)

| Attribute | |
|---|---|
| birthday | |
| last_name | |
| id | 0 |

# VISUALIZING THE HIERARCHY

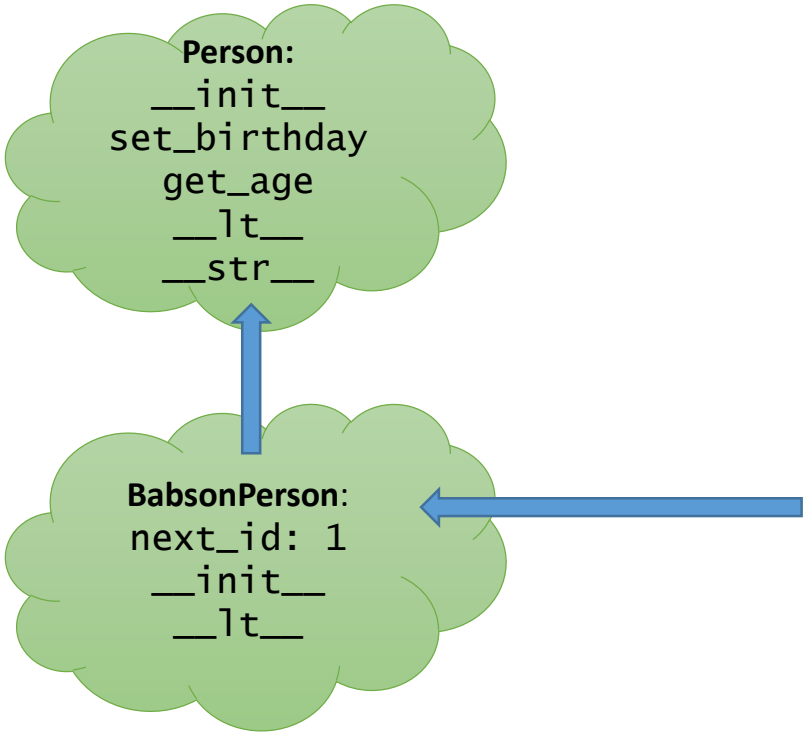



The rest of the original `__init__` code calls `self.id= BabsonPerson.next_id` which looks up `next_id` in the `BabsonPerson` environment, and creates a binding in `self` (i.e. the instance)

And then updates `next_id` in the `BabsonPerson` environment

| Attribute | |
|---|---|
| birthday | |
| last_name | |
| id | 0 |

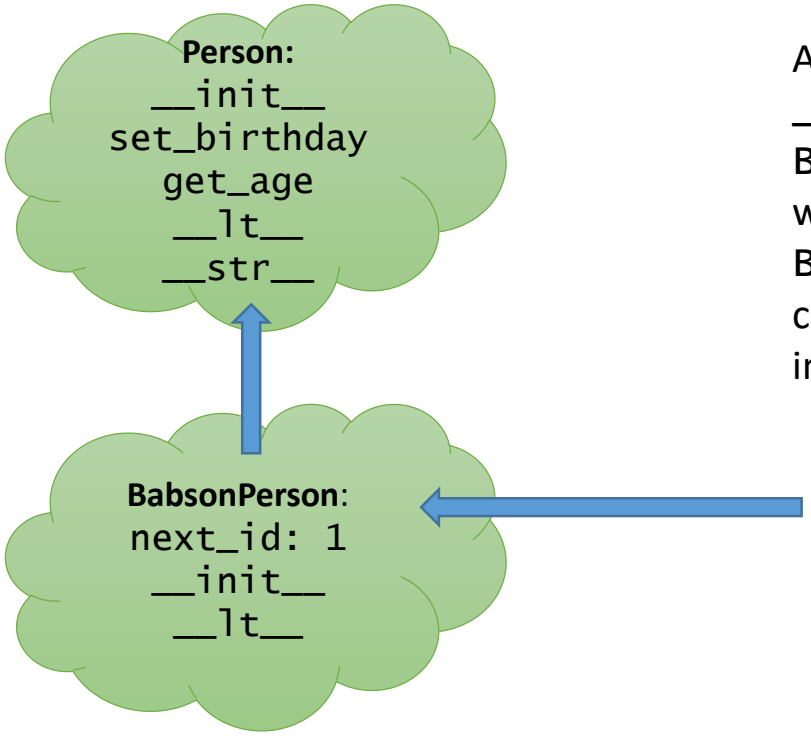
# VISUALIZING THE HIERARCHY




Thus calling `BabsonPerson` a second time to create a second instance will execute the same sequence, but now `next_id` is bound to 1

| Attribute | |
|---|---|
| birthday | |
| last_name | |

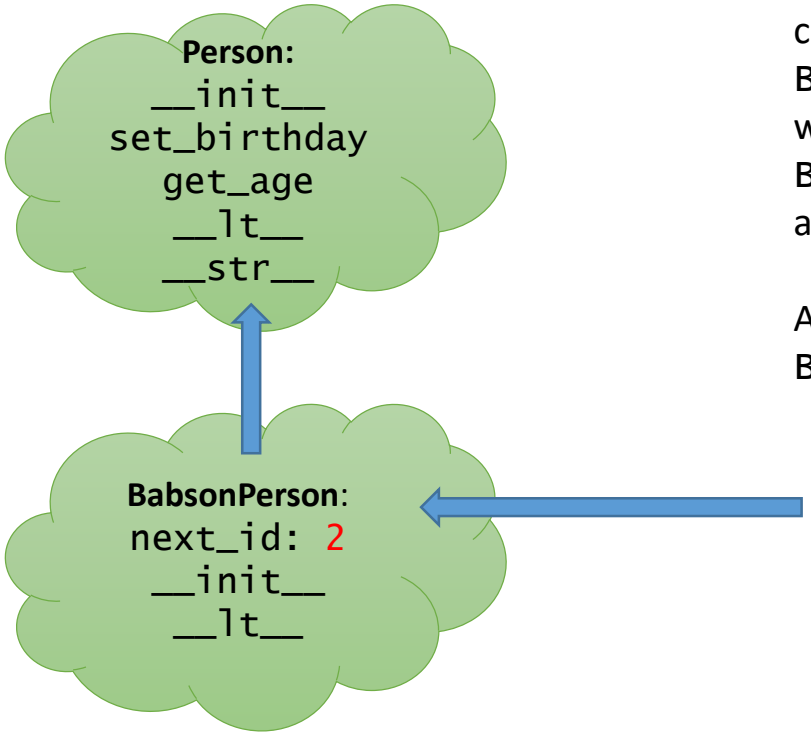
# VISUALIZING THE HIERARCHY




As before, the rest of the original `__init__` code calls `self.id= BabsonPerson.next_id` which looks up `next_id` in the `BabsonPerson` environment, and creates a binding in `self` (i.e. the instance)

| Attribute | |
|---|---|
| birthday | |
| last_name | |
| id | 1 |

# VISUALIZING THE HIERARCHY




The rest of the original `__init__` code calls `self.id= BabsonPerson.next_id` which looks up `next_id` in the `BabsonPerson` environment, and creates a binding in `self` (i.e. the instance)

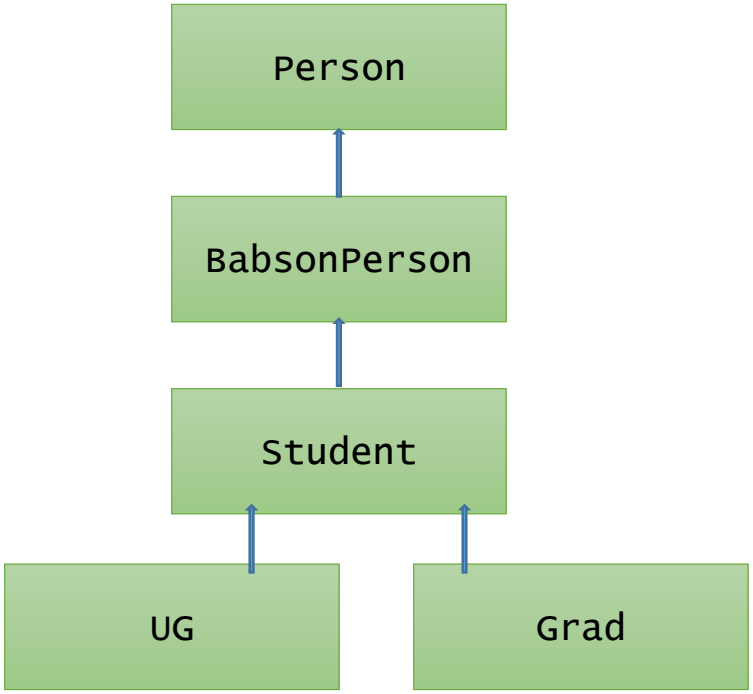
And then updates `next_id` in the `BabsonPerson` environment

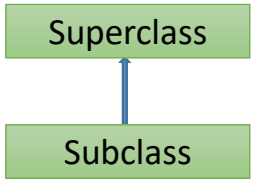
| Attribute | |
|---|---|
| birthday | |
| last_name | |
| id | 2 |

# USING INHERITANCE

- Explore in some detail an example of building an application that organizes information about people
  - `Person`: `name, birthday`
  - sort by last name
  - get age
- `BabsonPerson`: `Person` + ID Number
  - assign ID numbers in sequence
  - get ID number
  - sort by ID number
- `Students`: several types, all `BabsonPerson`
  - undergraduate student: has class year
  - graduate student

# CLASS HIERARCHY

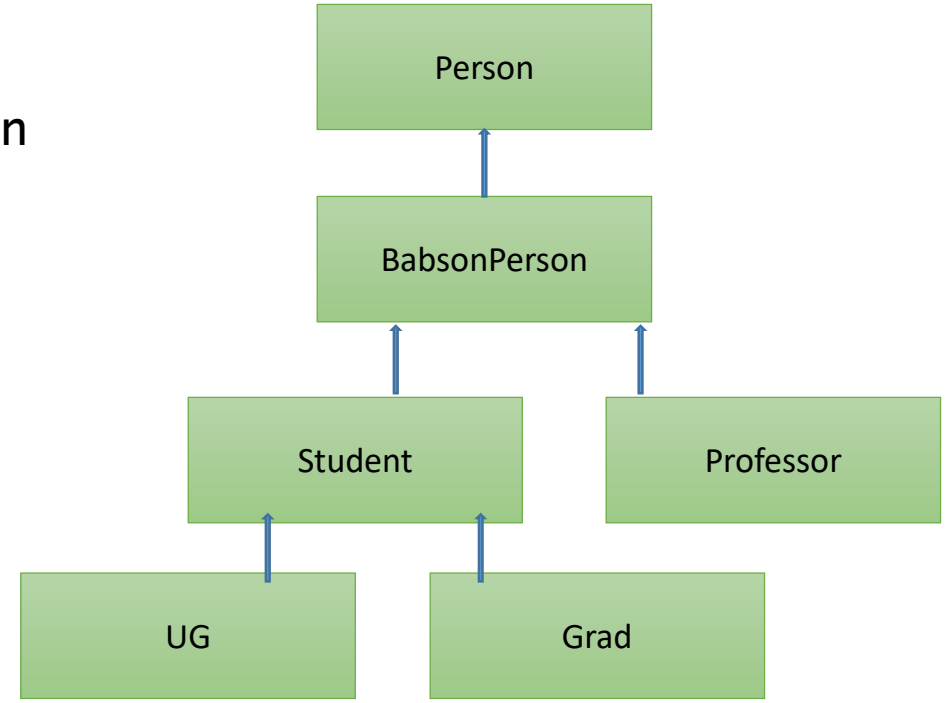


# EXERCISE

- Add a `Professor` Class:
  - Also a kind of `BabsonPerson`
  - May have different attributes and behaviors
  - Leverages existing methods from other classes in the hierarchy

- What is the benefit of modularity?



# ANOTHER CLASS: GRADES

- Create class, `Grades`, that includes instances of other classes within it
- Why?
  - build a data structure that can hold grades for students
  - gather together data and procedures for dealing with them in a single structure, so that users can manipulate without having to know internal details