Classes and Objects

CPSC 231: Introduction to Computer Science for Computer Science Majors I
Fall 2021

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 31 August 2021

UNIVERSITY OF CALGARY

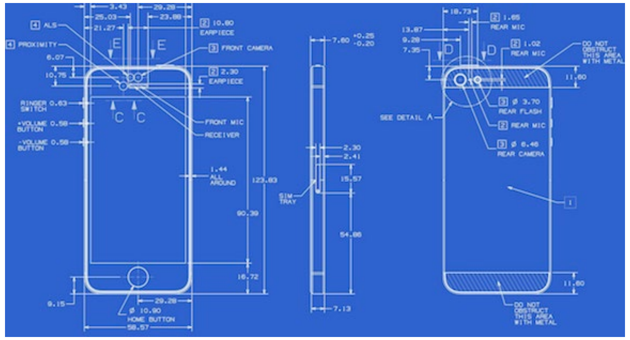
Class and Objects

Class

- A template that describes:
    - Fields (variables)
    - Methods (functions) operating on the data in the fields

Objects

- Instances of that class which take on different forms



Construction

Constructing an Object from a Class

- Variables of a class store pointers to objects (instances) of that class
- The process of creating an instance of an object is called instantiation/construction.
- Format:

```
<object name> = <name of the class> ()
```

- Example:

```
student1 = Student()
```

- The instantiation allocates memory space for the data fields and then associates the address with the object name

Fields

Classes

- A class is an abstract type that consists of **fields** and functions (**methods**) that operate on the data in the fields.
  - There are two types of fields
  1. **Class** fields (every object shares them)
  2. **Instance** fields (specific to one object)

Accessing fields

- Format:

```
<object name>.<field name>   # access an instance field
<object name>.<field name> = <value>  # change the value

<object type>.<field name>   # access a Class field
<object type>.<field name> = <value>  # change the value
```

- Example:

```
student1.name = 'Alice’

Student.MIN_ID = 1
```

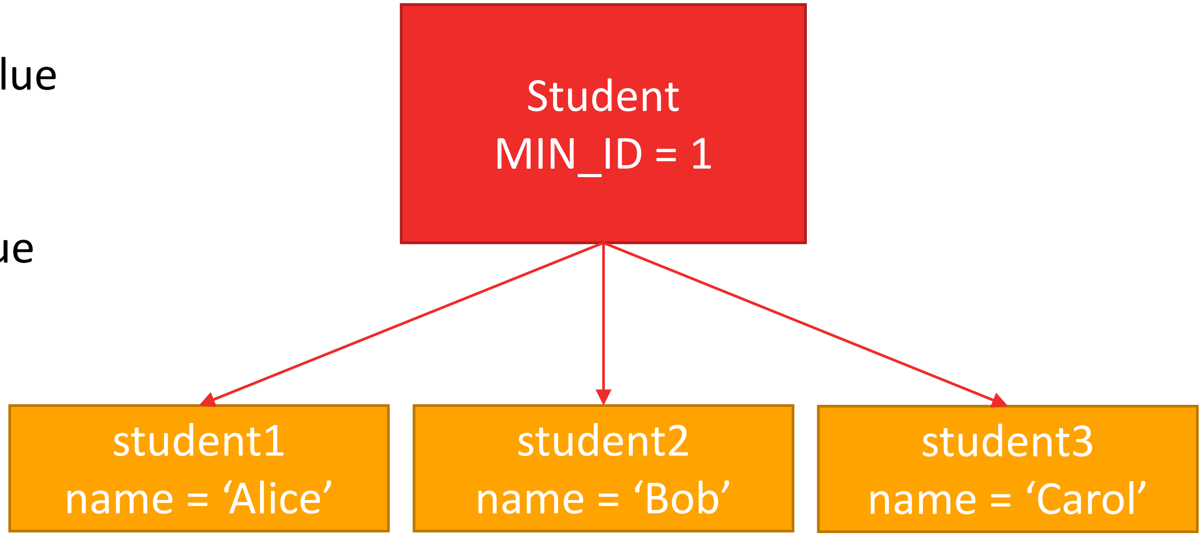
Accessing fields

- Format:

```
<object name>.<field name>   # access an instance field
<object name>.<field name> = <value>  # change the value

<object type>.<field name>   # access a Class field
<object type>.<field name> = <value>  # change the value
```

- Example:

```
student1.name = 'Alice’

Student.MIN_ID = 1
```



Each of the 3 instances has its own unique instance field values
All 3 share the Student class field values

Initializing the fields

- Class fields are initialized at as variables declared in the class itself
- Instance fields are **initialized** as variables within the constructor

```
class <name of the class>:
        <class field name>= <default value>

        def __init__(self, <param1>, …):
                self.<instance field name> = <param1>
```

Initialization

Initializing the attributes

- The constructor, a special method __init__(), is automatically called whenever an object is created and **initializes instance fields**.
- We can increase the complexity of this method based on how much we want to configure when an object is instantiated
- Format:

```
class Student:
    def __init__(self, first, last, address, phone, id):
        self.firstName = first
        self.lastName = last
        self.address = address
        self.phone = phone
        self.studentID = id
        self.courses = {}
```

Methods

Classes

- A class is an abstract that consists of fields and functions (**methods**) that operate on the data in the fields.
  - **Methods** act on the data from a class to transform it, update it, or retrieve it
- Format:

```
class <name of the class>:
        <class field name>= <default value>

        def __init__(self, <param1>, …):
                self.<instance field name> = <param1>

        def <method name> (self, <param1>, ...):
                method body
```

Simple Example

Classes

- Making a simple Student class
  - Class fields **MIN_ID, MAX_ID**
  - Instance fields **name**, **id**
  - One method -> prints out (**name**-**id**)

```
class Student:
        MAX_ID = 99999999
        MIN_ID = 0

        def __init__(self, new_name, new_id):
                self.name = new_name
                self.id = new_id

        def print(self):
                print(f”({self.name}-{self.id}”)
```

Objects

- Instance of a class (remember list(), set(), tuple())

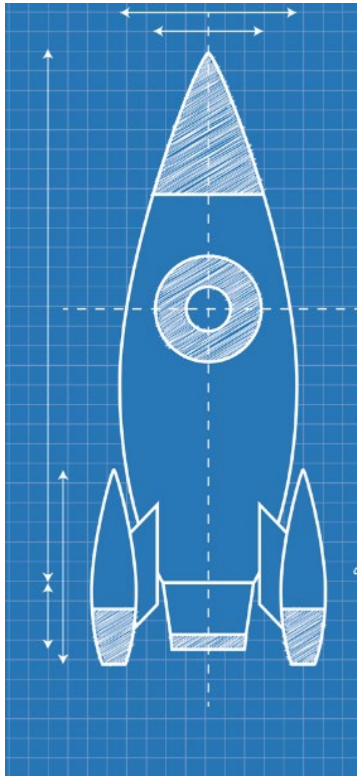
```
#Construct a student, automatically uses __init__(self, name, id)
student = Student(“Jonathan”, 999)
other = Student(“Dr.J”, 1)

#Print student info (Jonathan-999)
student.print()
#Print student info (Dr.J-1)
other.print()

#Access class field
print(Student.MIN_ID)

#Access instance field
print(student.name)
print(other.id)
```

Larger Example

Example

```
class Student:

    def __init__(self):
        self.lastName = ''
        self.firstName = ''
        self.studentID = 0
        self.address = ''
        self.phone = ''
        self.courses = {}

     # print the address of a student
    def printAddress (self):
        print(self.address)

     def addCourse (self, courseID):
        self.courses[courseID] = ""

    def assignGrade (self, courseID, grade):
        self.courses[courseID] = grade
```

This code does nothing!
It is just a blueprint.
A class description.



The **self** parameter is automatically set to reference the newly-created object that needs to be initialized.

Self?

What is self?

- The “self” reference allows an object to access its attributes inside its methods.
  - It is needed to distinguish the attributes of different objects of the same class.
  - **Object scope:** As long as the object is referenced by a name that is still active (valid), all of the attributes will be valid as well.

```
class Student:
        def __init__ (self,...):
                :

        def printInfo (self):
                :

# Main body
alice = Student(...)
jane = Student(...)
alice.printInfo()
jane.printInfo()
```

Motivating Complex Class Design

```
class Student:

    def __init__(self):
        self.lastName = ‘’
        self.firstName = ''
        self.studentID = 0
        self.address = ''
        self.phone = ''
        self.courses = {}

#Creating Alice the student
alice = Student()
alice.lastName = 'Smith'
alice.firstName = 'Alice'
alice.studentID = 12345678
alice.address = '55 Main Street'
alice.phone = '403-123-4567'
alice.courses[231] = 'A'
alice.courses[233] = 'B+'

print ('Name: %s %s' % (alice.firstName, alice.lastName))
print ('Student #: %d' % (alice.studentID))
print ('Address: %s' % (alice.address))
print ('Phone: %s' % (alice.phone))
print ('GPA: %s' % (alice.courses))
```

```
Name: Alice Smith
Student #: 12345678
Address: 55 Main Street
Phone: 403-123-4567
GPA: {231: 'A', 233: 'B+'}
```

```
class Student:

    def __init__(self, lname=“, fname=“, id=0, add=“, ph=“):
        self.lastName =  lname
        self. firstName = fname
        self. studentID = id
        self. address = add
        self. phone = ph
        self. courses = {}

#Creating Alice the student
alice = Student(‘Smith’,’Alice’,12345678,’55 Main Street’,’403-123-4567’)
alice.courses[231] = 'A'
alice.courses[233] = 'B+'

print ('Name: %s %s' % (alice.firstName, alice.lastName))
print ('Student #: %d' % (alice.studentID))
print ('Address: %s' % (alice.address))
print ('Phone: %s' % (alice.phone))
print ('GPA: %s' % (alice.courses))
```

```
Name: Alice Smith
Student #: 12345678
Address: 55 Main Street
Phone: 403-123-4567
GPA: {231: 'A', 233: 'B+'}
```

```
class Student:

    def __init__(self, lname=“, fname=“, id=0, add=“, ph=“):
        self.lastName =  lname
        self. firstName = fname
        self. studentID = id
        self. address = add
        self. phone = ph
        self. courses = {}
    def print(self):
        print ('Name: %s %s' % (self.firstName, self.lastName))
        print ('Student #: %d' % (self.studentID))
        print ('Address: %s' % (self.address))
        print ('Phone: %s' % (self.phone))
        print ('GPA: %s' % (self.courses))

#Creating Alice the student
alice = Student(‘Smith’,’Alice’,12345678,’55 Main Street’,’403-123-4567’)
alice.courses[231] = 'A'
alice.courses[233] = 'B+’
alice.print()
```

```
Name: Alice Smith
Student #: 12345678
Address: 55 Main Street
Phone: 403-123-4567
GPA: {231: 'A', 233: 'B+'}
```

Changing Data: Methods

Methods in Class

- Class methods are used to
  - hide the implementation detail
    - e.g., addCourse() and assignGrade() allows to change course information without knowing its implementation
- Provide common methods to be used by the objects
  - e.g., printAddress(), printInfo())
- A class method is just like a regular function

```
class Student:
    def__init__(self):
        self.lastName = ‘’
        self.firstName = ‘’
        self.studentID = 0
        self.address = ''
        self.phone = ''
        self.courses = {}

    def printInfo (self):
        print ('Name: %s %s' % (self.firstName, self.lastName))
        print ('Student #: %d' % (self.studentID))
        print ('Address: %s' % (self.address))
        print ('Phone: %s' % (self.phone))
        print ('GPA: %s' % (self.courses))
        print ()

    def addCourse (self, courseID):
        self.courses[courseID] = ''

    def assignGrade (self, courseID, grade):
        self.courses[courseID] = grade
```

```
# The main body of the program
alice = Student()
alice.lastName = 'Smith'
#...
alice.addCourse(231)       # add 231 to the course list
alice.addCourse(233)       # add 233 to the course list
alice.assignGrade(231, 'A')  # assign grade for 231
alice.assignGrade(233, 'B+')  # assign grade for 233
alice.printInfo()

# Create another student
jane = Student()
#...
jane.printInfo()
```

```
Name: Alice Smith
Student #: 12345678
Address: 55 Main Street
Phone: 403-123-4567
GPA: {231: 'A', 233: 'B+'}
```

Why classes?

Why classes?

- Using classes allows new types of variables to be declared
  - The new type can model information about any arbitrary entity (e.g., car, movie, pet, you name it)
- A predetermined number of fields can be specified in the class definition and those fields can be named
- Hiding information and creating interface (through methods) so that changes inside a class has minimal impact on the rest of the program
- Organizing the code makes it scalable and easier to maintain

Try!

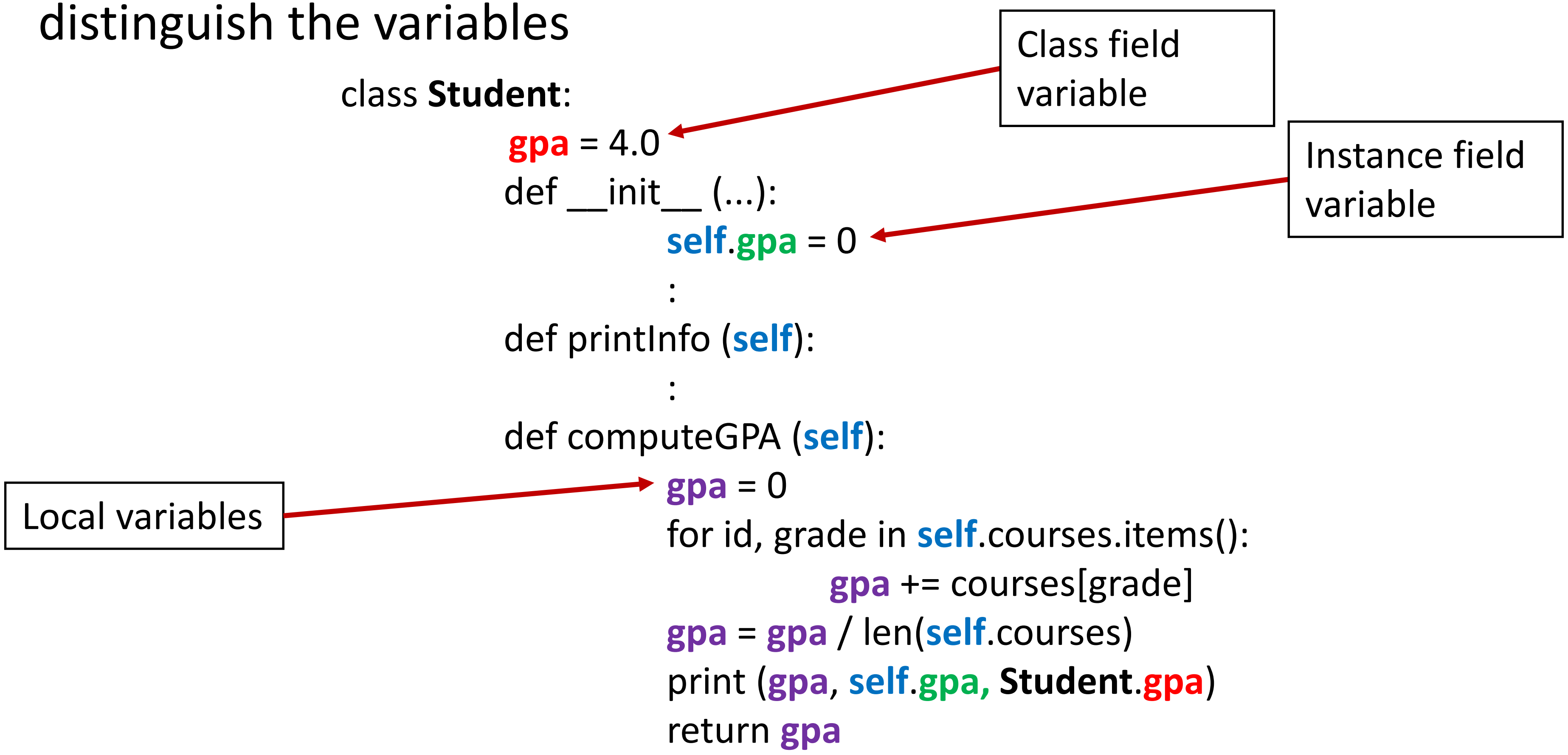
Practice

- Create a class for a pet!

Accessing

Accessing attributes and methods

- A function may have a local variable with the same name as a instance field variable or a class field variable, the keyword “self” or <class name> is used to distinguish the variables



```
class Student:
    gpa = 4.0
    def __init__ (...):
        self.gpa = 0
        :
    def printInfo (self):
        :
    def computeGPA (self):
        gpa = 0
        for id, grade in self.courses.items():
            gpa += courses[grade]
        gpa = gpa / len(self.courses)
        print (gpa, self.gpa, Student.gpa)
        return gpa
```

Class field variable

Instance field variable

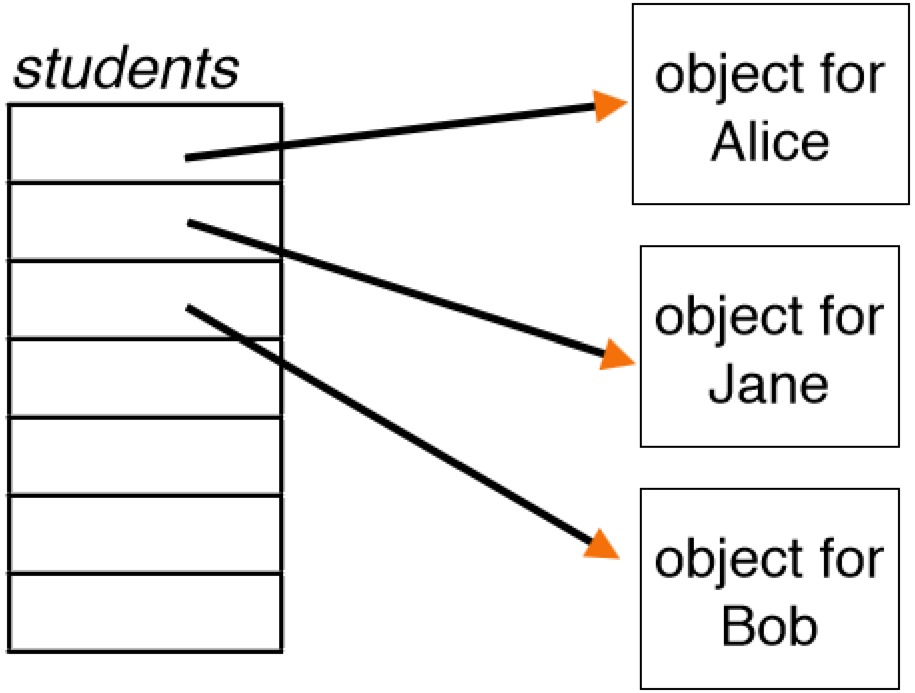
Local variables

Lists of Objects

Lists of objects

```
students = []
 :
students.append(student)
```

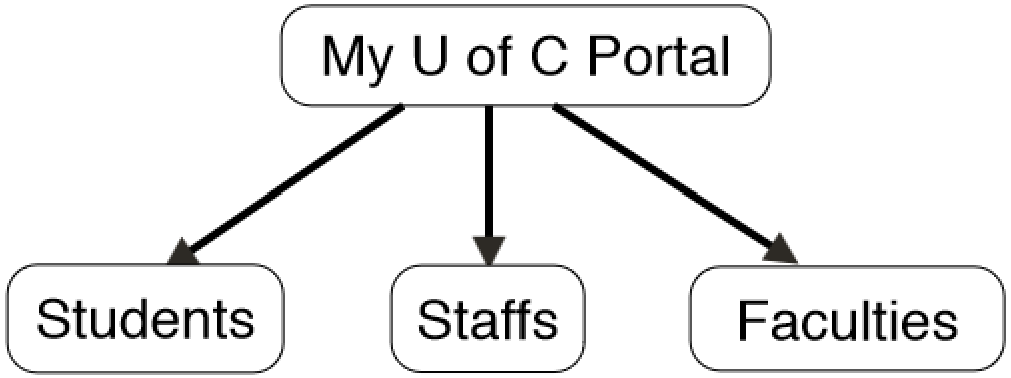
- Each element in the list is a reference to an object



Design

Class design

- So far we decomposed problems into small tasks and implemented each using a function.
- To incorporate classes into the design of your solution:
  - We look at the data and their logical relationships
  - We then decide on the methods needed for each data set



Class design

- The class design is like a black box, which has a known input and output, but how it works is a mystery.
  - A class should maintains certain information and performs a known set of operations.
  - The actual implementation is irrelevant to the rest of the program as long as the parameter lists of each class method remains unchanged.
- Such separation of the class implementation from the rest of the program is called **encapsulation**.

Module System

Modules

- A large program may contain tens (if not hundreds or thousands) of classes. Instead of managing the entire program in a single file, Python allows us to divide the program into parts
  - Each part is a module contained in a separate file where the file name is the same as the module name.
  - In order to access a module, we must "import" it.
  - Format:

```
from <file name> import <function or class name>
OR
import <file name>
```

Modules

- A large program contains thousands of lines of code
- Python allows to divide the program into parts
- Each part is a module contained in a separate file named the same as the module name.
- In order to access a module, we must “import” it.

Hello.py

```
def helloFunc():
        print ("Hello World!")
```

Goodbye.py

```
def goodbyeFunc(name):
        print("Goodbye", name )
```

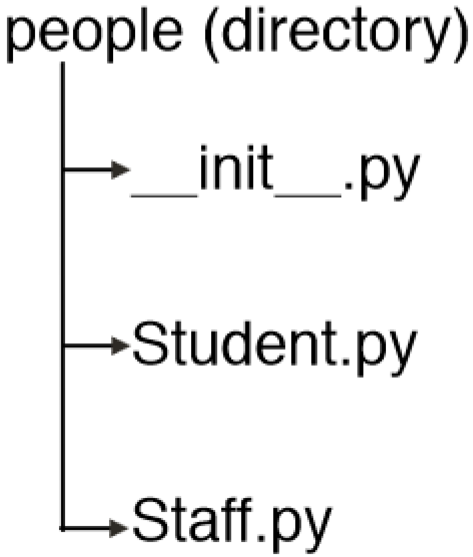
Main.py

```
import Hello
from Goodbye import *
def main():
        Hello.helloFunc()
        goodbyeFunc(“Classmate")

main()
```

Packages

- In Python, packages use the structure of the directories to make many files in the same directory accessible like a single module
- To create and use a package:
  - Create a directory with the name of the package (e.g., people)
  - In the directory, have each class in a separate *.py file (e.g., Student.py and Staff.py), where the file names match the class names.
  - In the same directory, create a file called __init__.py
  - This file tells Python that this is a package directory, and not just a directory with Python files in it.
  - In this file, import each module within this package
  - In the main program, import the package (e.g., import People)

Example



```
from People.Student import Student
from People.Staff import Staff
```

```
class Student:
     :
```

```
class Staff:
     :
```

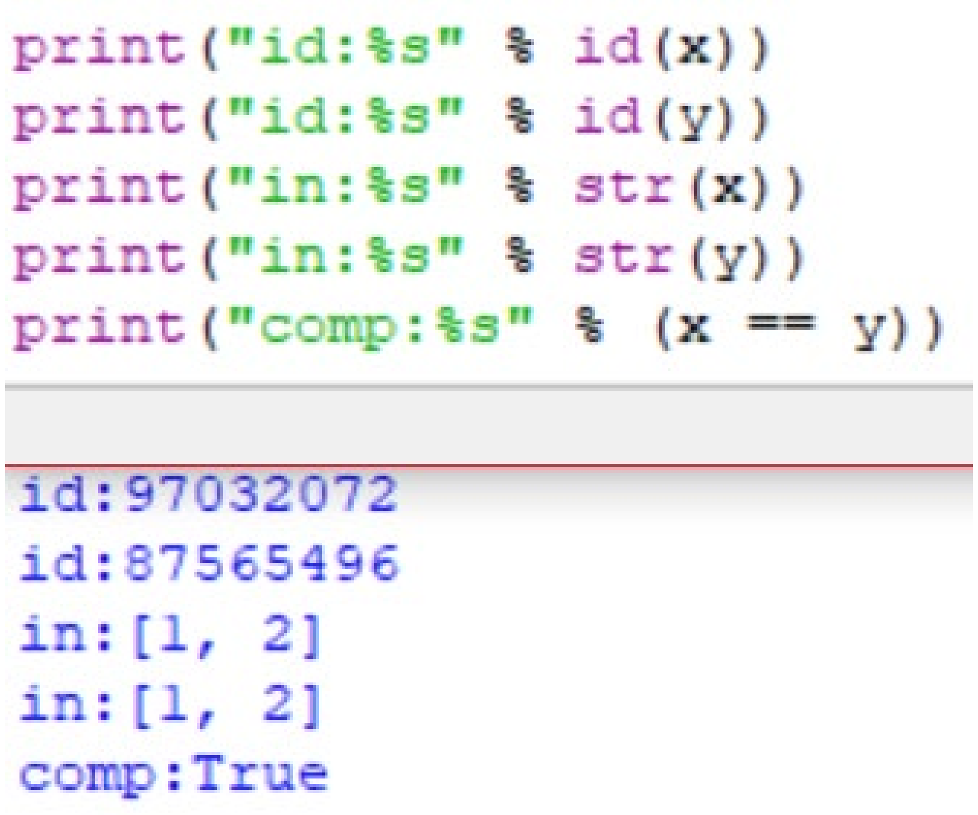
main.py

```
from People import *
     :
     student = Student(...)
     :
     staff = Staff(...)
```

Identity/Equality

Classes and identity

- Every class (data structure you make has an internal python identity)

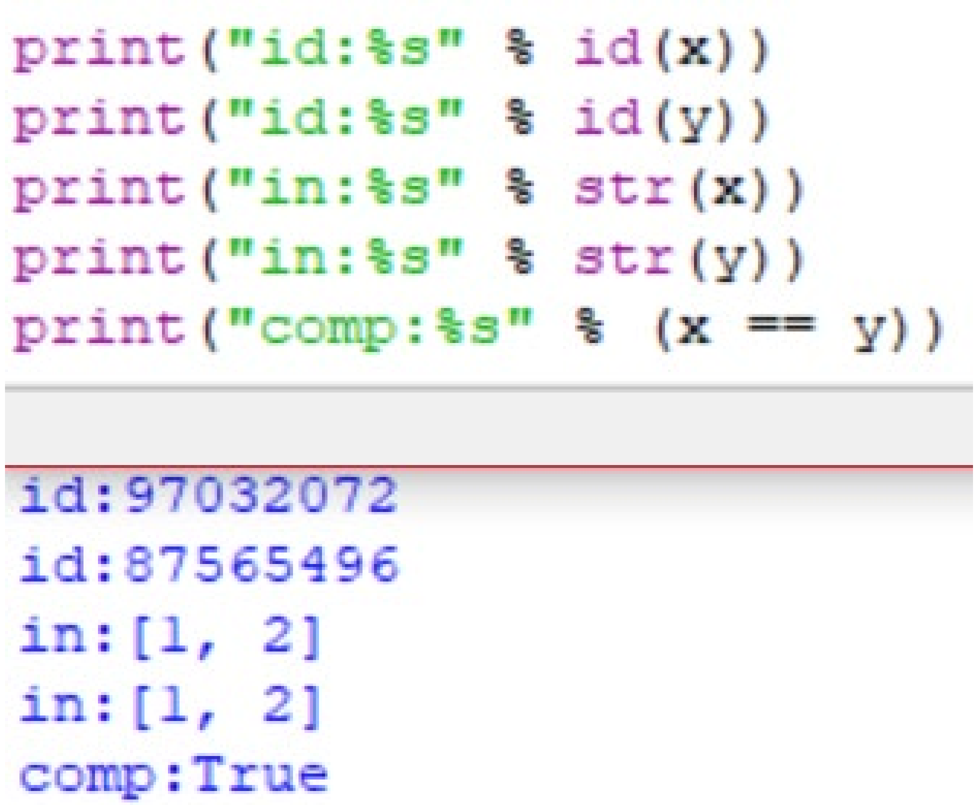


```
print("id:%s" % id(x))
print("id:%s" % id(y))
print("in:%s" % str(x))
print("in:%s" % str(y))
print("comp:%s" % (x == y))
```

```
id:97032072
id:87565496
in:[1, 2]
in:[1, 2]
comp:True
```

Classes and identity

- You'll have noticed that python knows how to sort strings, print the data structure, or compare contents on existing data structures



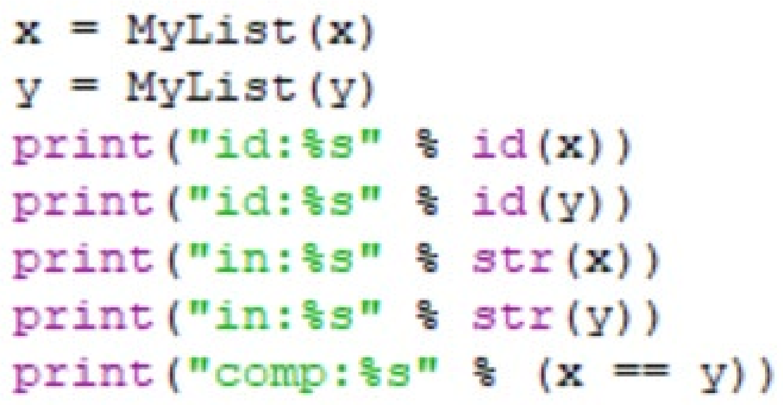
```
print("id:%s" % id(x))
print("id:%s" % id(y))
print("in:%s" % str(x))
print("in:%s" % str(y))
print("comp:%s" % (x == y))
```

```
id:97032072
id:87565496
in:[1, 2]
in:[1, 2]
comp:True
```

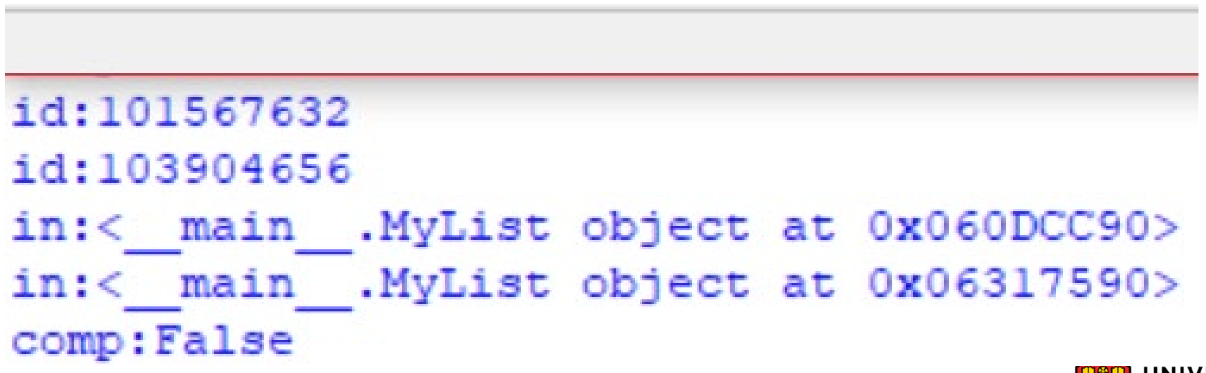
Classes and identity

- **But you'll notice yours operate differently at first!!!!**

```
class MyList:
    def __init__(self, new_list):
        self.my_list = new_list
```



```
x = MyList(x)
y = MyList(y)
print("id:%s" % id(x))
print("id:%s" % id(y))
print("in:%s" % str(x))
print("in:%s" % str(y))
print("comp:%s" % (x == y))
```

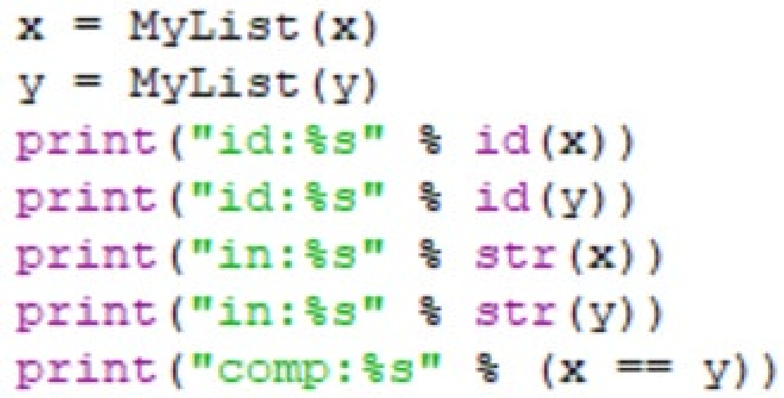


```
id:101567632
id:103904656
in:<__main__.MyList object at 0x060DCC90>
in:<__main__.MyList object at 0x06317590>
comp:False
```

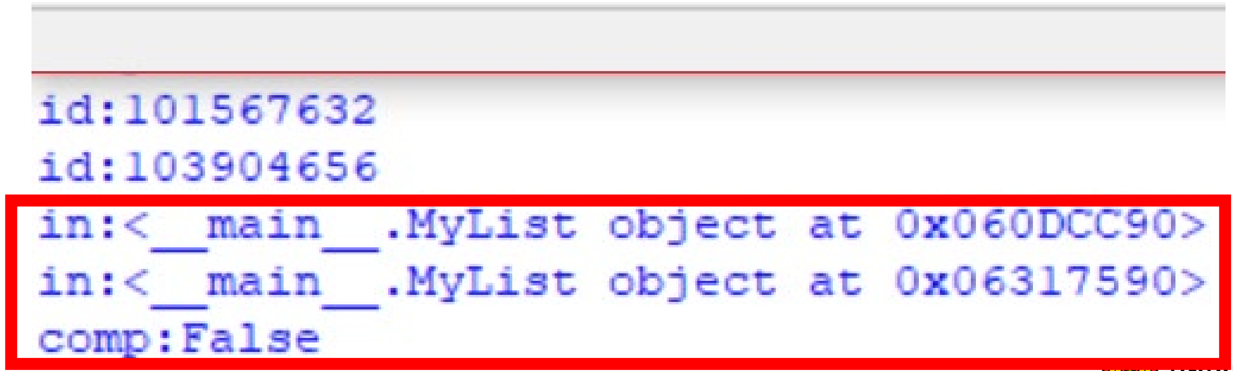
Classes and identity

- **But you’ll notice yours operate differently at first!!!!**

```
class MyList:
    def __init__(self, new_list):
        self.my_list = new_list
```



```
x = MyList(x)
y = MyList(y)
print("id:%s" % id(x))
print("id:%s" % id(y))
print("in:%s" % str(x))
print("in:%s" % str(y))
print("comp:%s" % (x == y))
```



```
id:101567632
id:103904656
in:<__main__.MyList object at 0x060DCC90>
in:<__main__.MyList object at 0x06317590>
comp:False
```

Classes and identity

- Three key concepts that exist
  - How to compare (how to hash/equality)?
  - How to print?
  - How to order?

Classes and identity

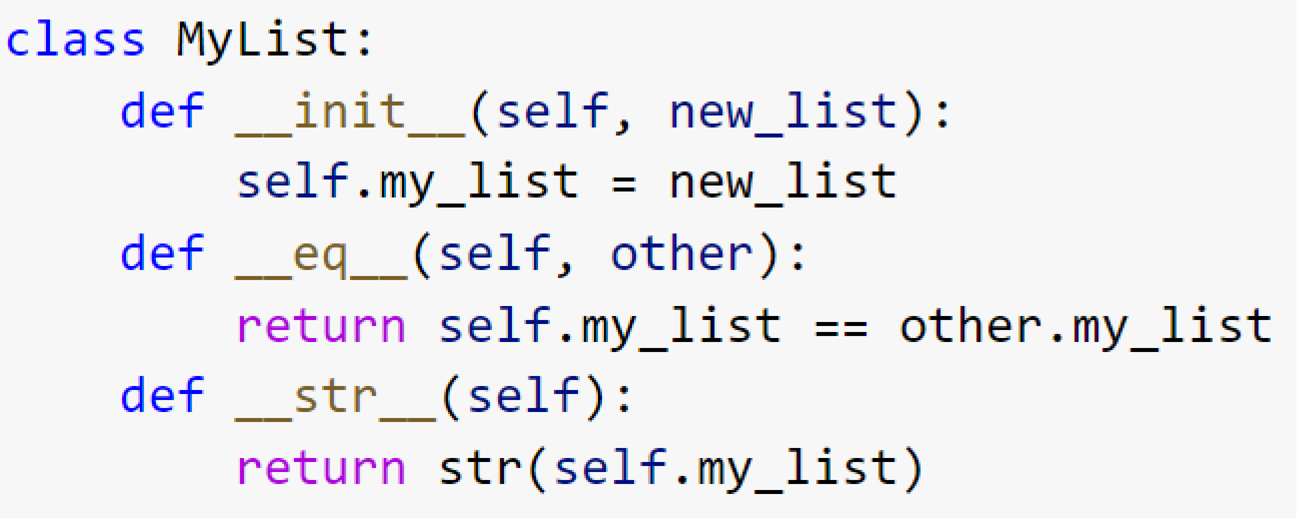
- Three key concepts that exist
  - How to compare (how to hash/equality)? __eq__(self, other) __hash__(self)
  - How to print? __str__(self)
  - How to order? __lt__(self)

Classes and identity

- Three key concepts that exist
  - How to compare (how to hash/**equality**)? **__eq__(self, other)** __hash__(self)
  - How to **print**? **__str__(self)**
  - How to order? __lt__(self)

Classes and identity

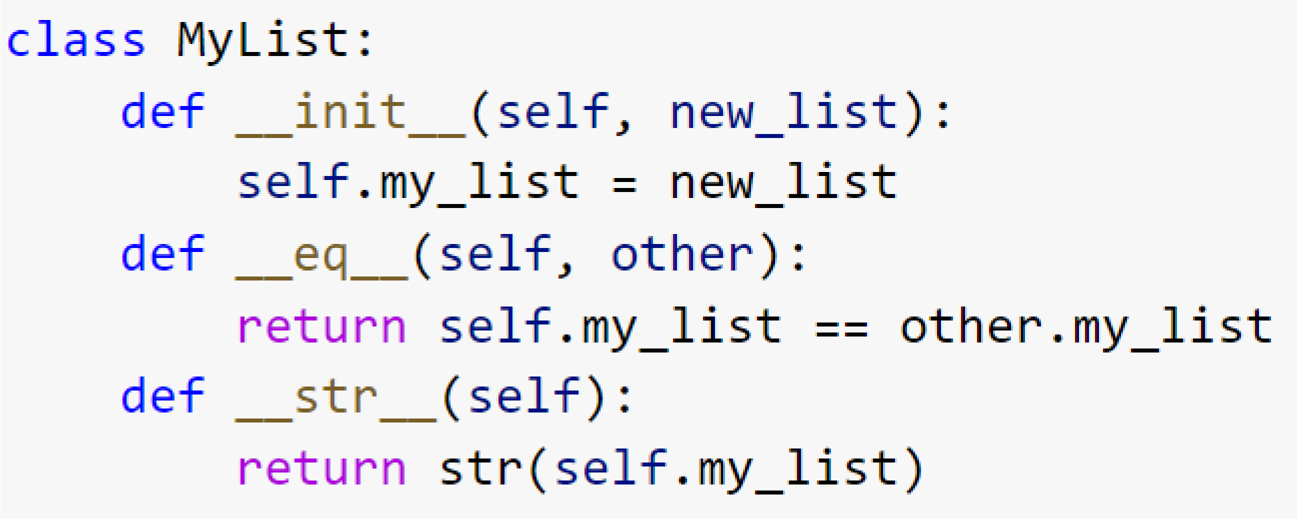
- Three key concepts that exist
  - How to compare (how to hash/**equality**)? **__eq__(self, other)** __hash__(self)
  - How to **print**? **__str__(self)**
  - How to order? __lt__(self)



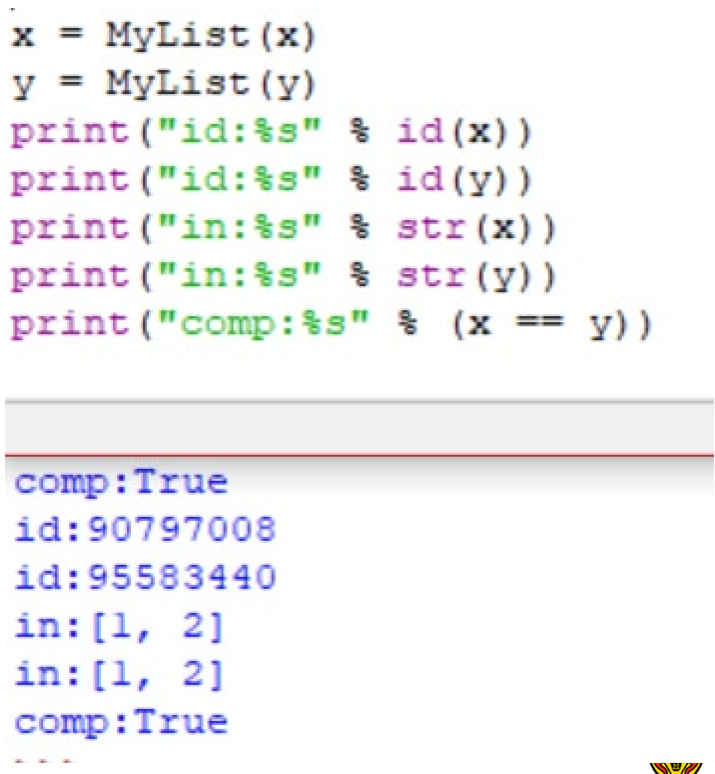
```
class MyList:
    def __init__(self, new_list):
        self.my_list = new_list
    def __eq__(self, other):
        return self.my_list == other.my_list
    def __str__(self):
        return str(self.my_list)
```

Classes and identity

- Three key concepts that exist
  - How to compare (how to hash/**equality**)? **__eq__(self, other)** __hash__(self)
  - How to **print**? **__str__(self)**
  - How to order? __lt__(self)



```python
class MyList:
    def __init__(self, new_list):
        self.my_list = new_list
    def __eq__(self, other):
        return self.my_list == other.my_list
    def __str__(self):
        return str(self.my_list)
```



```python
x = MyList(x)
y = MyList(y)
print("id:%s" % id(x))
print("id:%s" % id(y))
print("in:%s" % str(x))
print("in:%s" % str(y))
print("comp:%s" % (x == y))
```

```
comp:True
id:90797008
id:95583440
in:[1, 2]
in:[1, 2]
comp:True
```

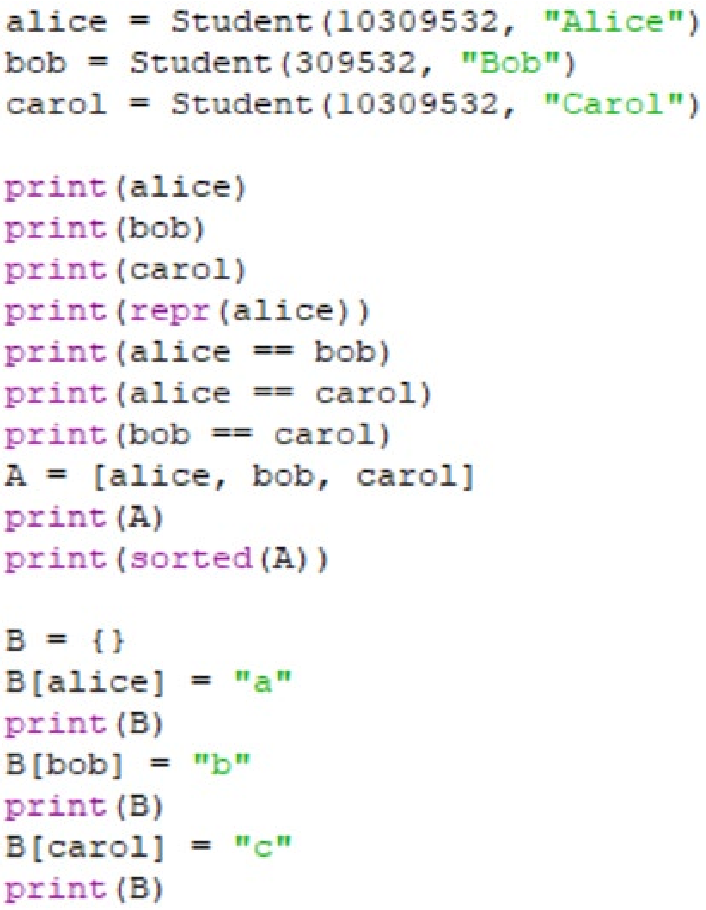
Ordering/Hashing

What about ordering and hashing? Student Example

```python
class Student:
    def __init__(self, sid, name):
        self.sid = sid
        self.name = name
    def __str__(self):
        return "(%s, %s)" % (self.sid, self.name)
    def __repr__(self):
        return "Student(%s, %s)" % (self.sid, self.name)
    def __eq__(self, other):
        return self.sid == other.sid
    def __lt__(self, other):
        if self.sid < other.sid:
            return True
        elif self.sid > other.sid:
            return False
        if self.name < other.name:
            return True
        return False
    def __hash__(self):
        return hash(self.sid)
```

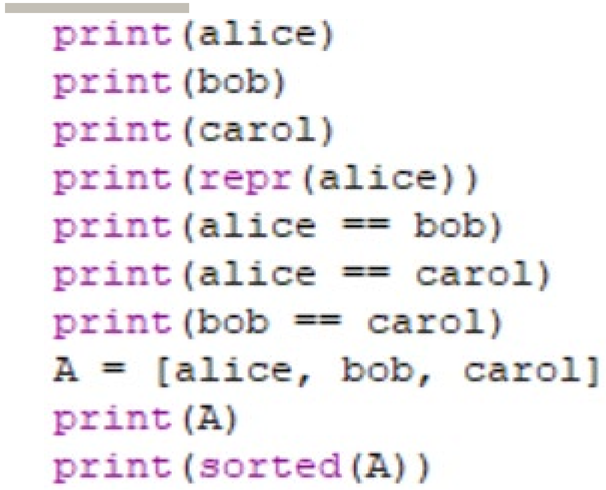
What about ordering and hashing? Student Example

```
class Student:
    def __init__(self, sid, name):
        self.sid = sid
        self.name = name
    def __str__(self):
        return "(%s, %s)" % (self.sid, self.name)
    def __repr__(self):
        return "Student(%s, %s)" % (self.sid, self.name)
    def __eq__(self, other):
        return self.sid == other.sid
    def __lt__(self, other):
        if self.sid < other.sid:
            return True
        elif self.sid > other.sid:
            return False
        if self.name < other.name:
            return True
        return False
    def __hash__(self):
        return hash(self.sid)
```

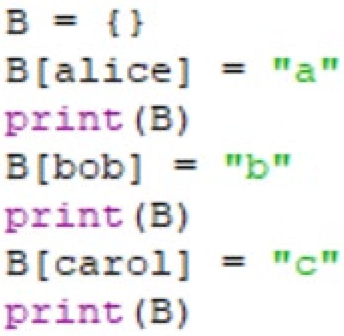


```
alice = Student(10309532, "Alice")
bob = Student(309532, "Bob")
carol = Student(10309532, "Carol")

print(alice)
print(bob)
print(carol)
print(repr(alice))
print(alice == bob)
print(alice == carol)
print(bob == carol)
A = [alice, bob, carol]
print(A)
print(sorted(A))

B = {}
B[alice] = "a"
print(B)
B[bob] = "b"
print(B)
B[carol] = "c"
print(B)
```

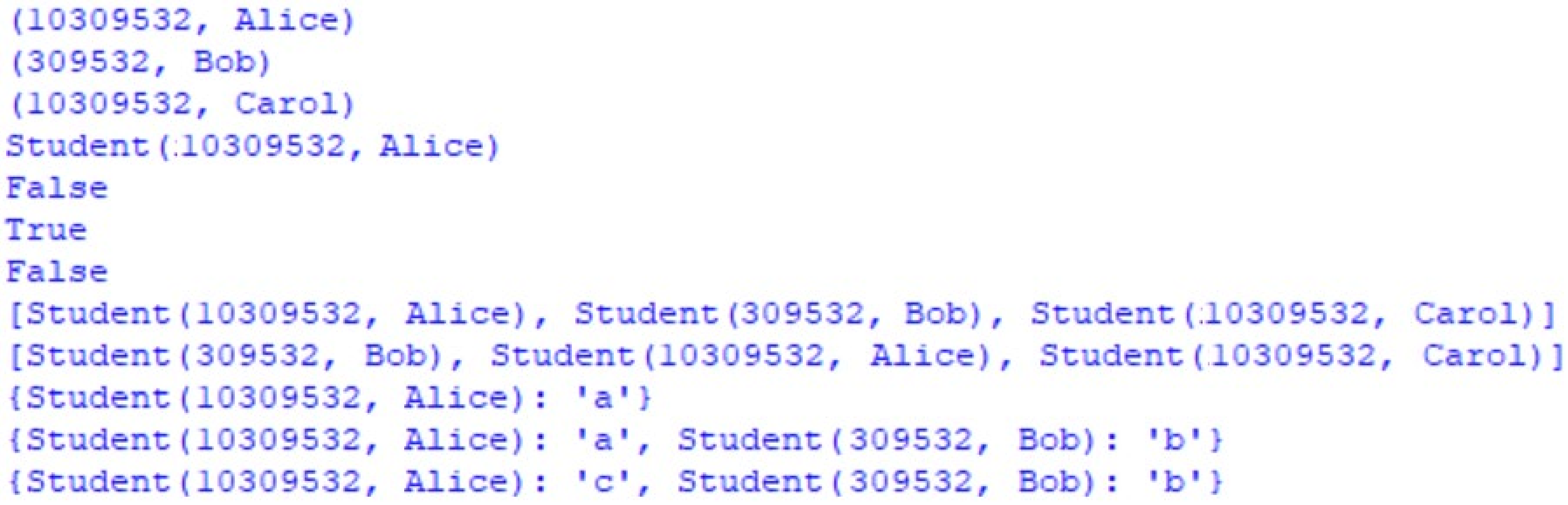
What about ordering and hashing? Student Example



```
print(alice)
print(bob)
print(carol)
print(repr(alice))
print(alice == bob)
print(alice == carol)
print(bob == carol)
A = [alice, bob, carol]
print(A)
print(sorted(A))
```



```
B = {}
B[alice] = "a"
print(B)
B[bob] = "b"
print(B)
B[carol] = "c"
print(B)
```

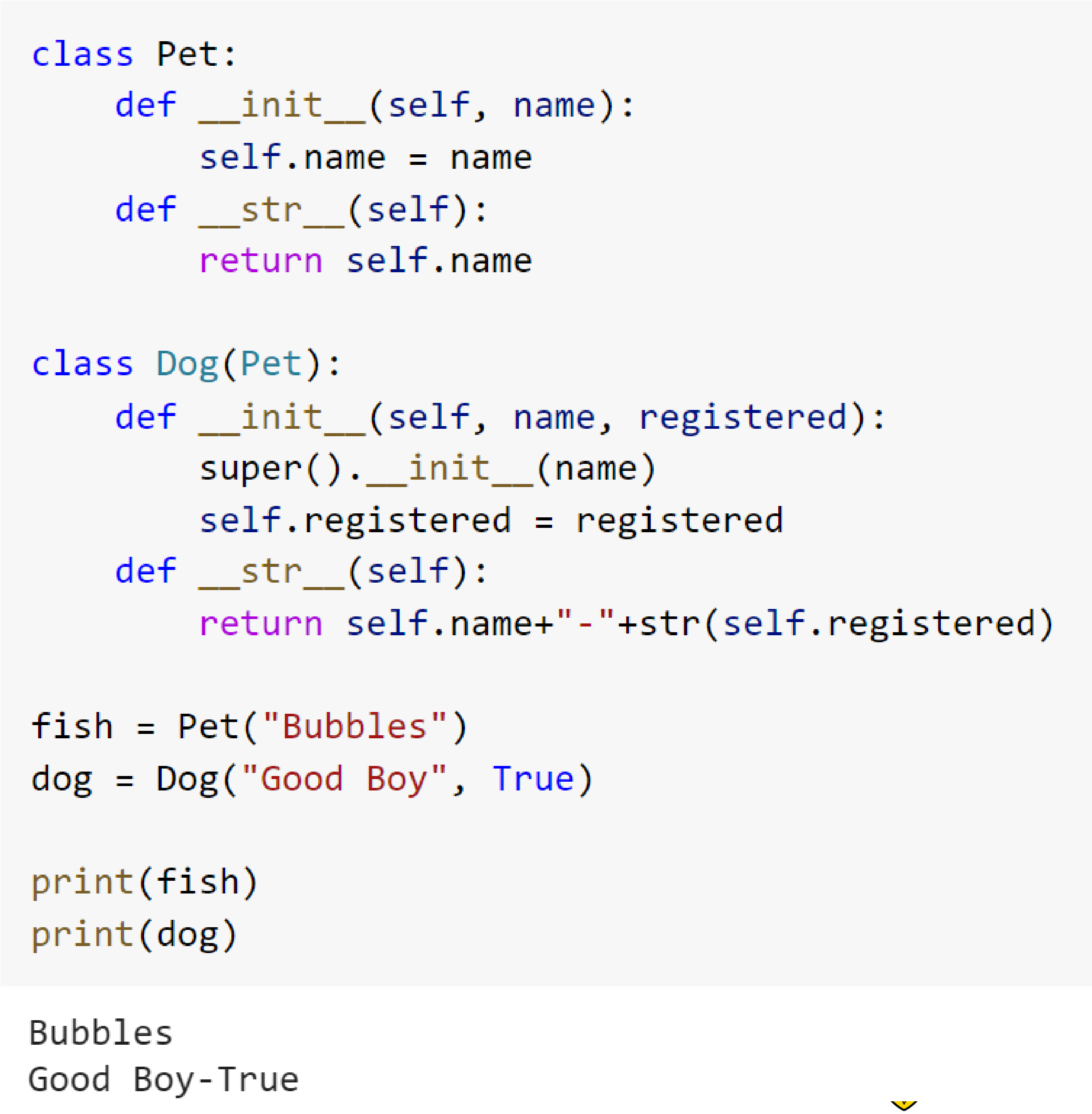


```
(10309532, Alice)
(309532, Bob)
(10309532, Carol)
Student(10309532, Alice)
False
True
False
[Student(10309532, Alice), Student(309532, Bob), Student(10309532, Carol)]
[Student(309532, Bob), Student(10309532, Alice), Student(10309532, Carol)]
{Student(10309532, Alice): 'a'}
{Student(10309532, Alice): 'a', Student(309532, Bob): 'b'}
{Student(10309532, Alice): 'c', Student(309532, Bob): 'b'}
```

Inheritance

Inheritance

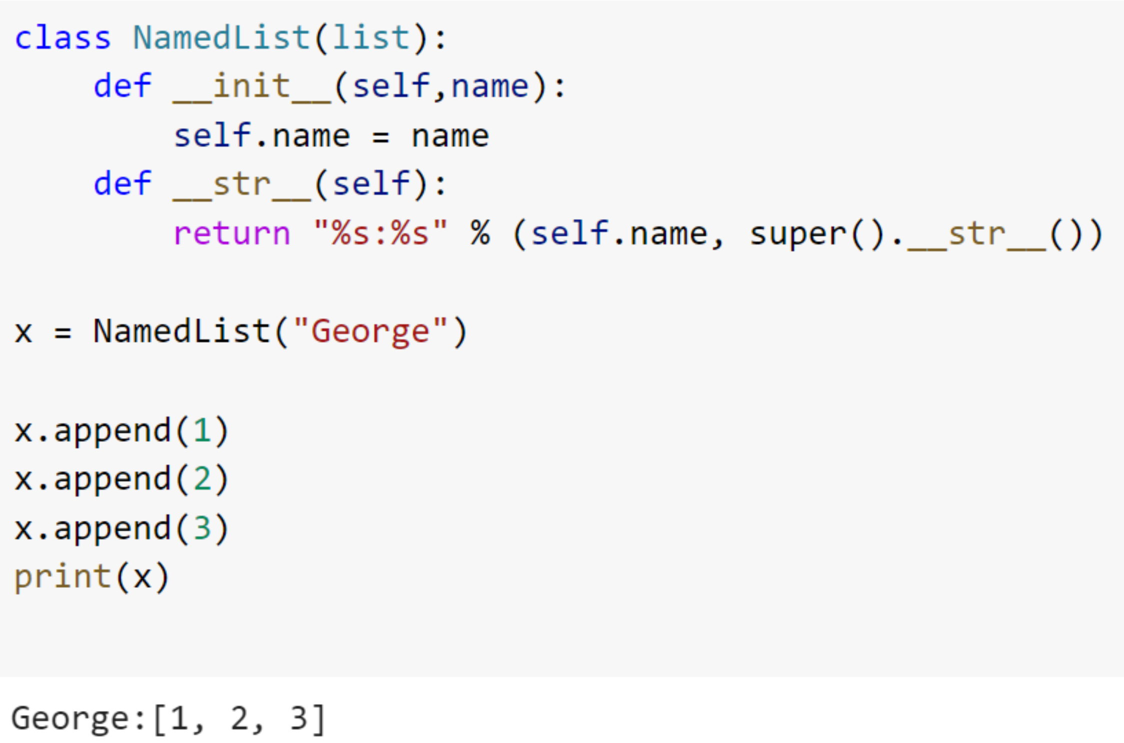
- You can make classes that gain properties of other classes
- Here Dog is a sub-class of Pet
- Pet is the super-class of Dog
- Dogs can be registered with the city
- Both can use the string method from Pet to print them using their name



```python
class Pet:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name

class Dog(Pet):
    def __init__(self, name, registered):
        super().__init__(name)
        self.registered = registered
    def __str__(self):
        return self.name+"-"+str(self.registered)

fish = Pet("Bubbles")
dog = Dog("Good Boy", True)

print(fish)
print(dog)
```

```
Bubbles
Good Boy-True
```

Inheritance



```
class NamedList(list):
    def __init__(self,name):
        self.name = name
    def __str__(self):
        return "%s:%s" % (self.name, super().__str__())

x = NamedList("George")

x.append(1)
x.append(2)
x.append(3)
print(x)
```

```
George:[1, 2, 3]
```

We can also extend python class, here I made a version of the list class that also stores a name for every list, I get for free everything the list did before

Onward to … recursion.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY