http://www.cs.cornell.edu/courses/cs1110/2022sp

Lecture 15:
Classes
(Chapters 15 & 17.1-17.5)

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Lecture Afterthoughts

- Call Frame on slide 10 is new. Check it out!
- Slide 27 had a typo! Needed to create the Course before we could enroll in it
- The lecture stopped at slide 29 but slides 30-37 are also worth taking a peek at (including a Q&A)

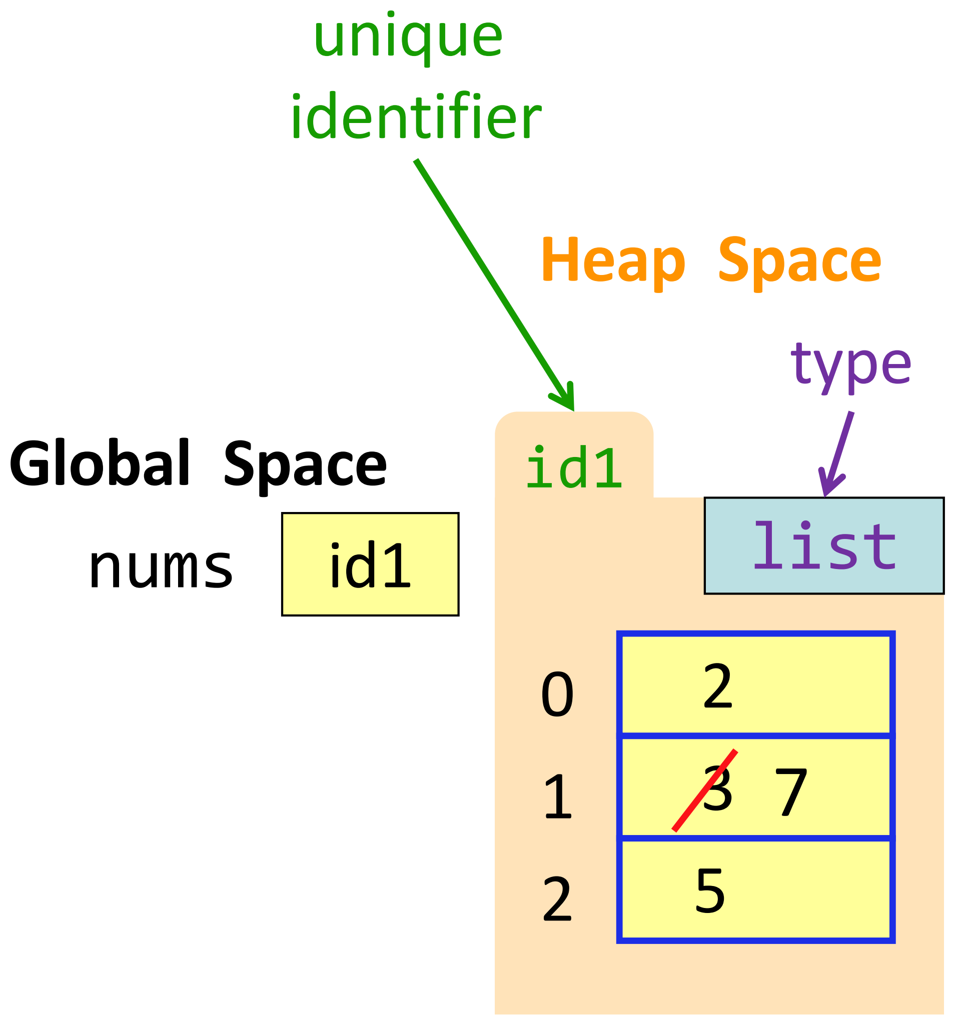
Announcements

- Prelim 2 alternate time request form live Fri 3/25
- More 1-on-1's today thru Sunday.
  - Come one, come all! (Sign up on CMS.)
- A5 due date moved later to Sun 4/17.
  - The tradeoff: more time to work on A5, less "pressure" on Spring break, **BUT** less time to look at the A5 solutions before Prelim 2 (Tu 4/19) and temptation to delay prelim studying. (Resist that temptation.)
- next week's lab 16 extended to Wed 4/13 due to spring break
- These updates are on the Schedule webpage.

Recall: Objects as Data in Folders

- attributes: variables within objects
- Type shown in the corner

```
nums = [2,3,5]
nums[1] = 7
```



unique identifier

Heap Space

type

Global Space

nums id1

id1

list

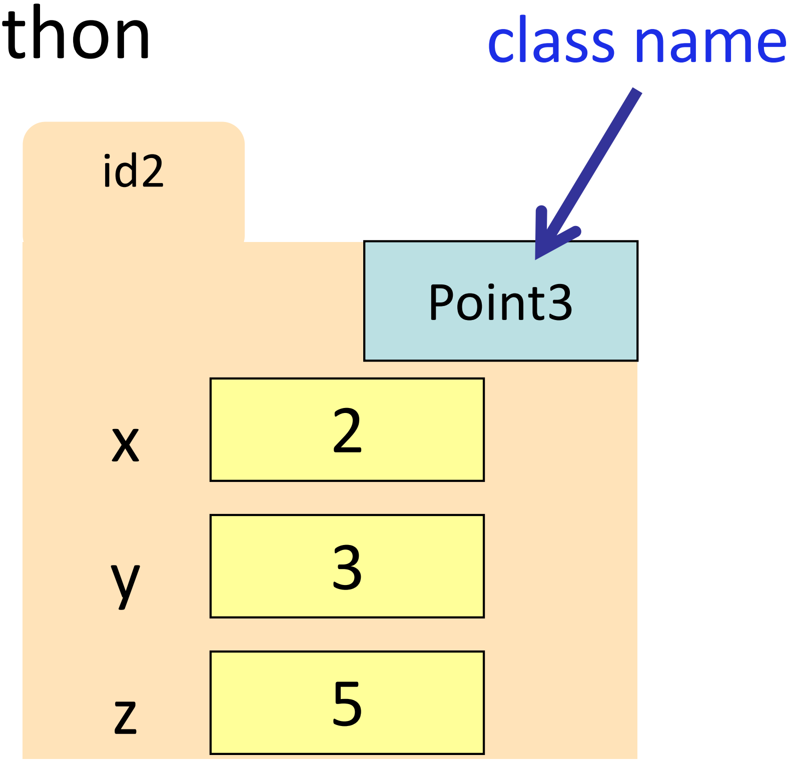
0 2

1 ~~3~~ 7

2 5

Classes are user-defined Types

Defining new classes = adding new types to Python



Example Classes

- Point3
- Rect
- Freq (A3), for word frequencies
- Doll (class, lab)
- Song, Mix (A4)

Simple Class Definition

```
class <class-name>:
   """Class specification"""
   <method definitions>
```

Just like function definitions, but placed inside a class definition, i.e., indented relative to the class header

The Class Specification

```
class Course:
    """An instance is a Cornell course

    Instance Attributes:
    name:      [str] name of the course of form: <DEPT NUM>
    n_credit:  [int] number of credits, must be > 0
    """
```

Short Summary

Attribute list

Attribute name

*Description and invariant**

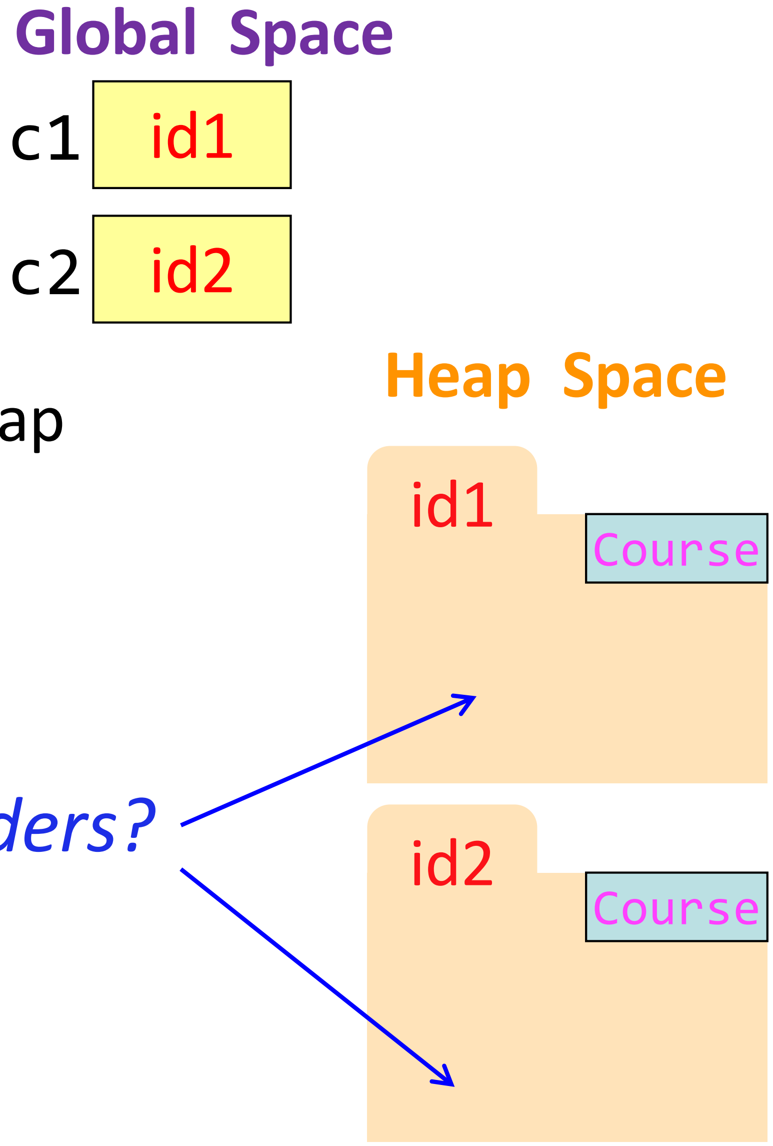
**more about this later in this lecture*

Convention: capitalize first letter of class name

Constructor (1)

- Function to create new instances
  - function name is the class name
- Calling the constructor:
  - Makes a new object (folder) on the Heap
  - Returns the id of the folder

But how do we populate the folders?

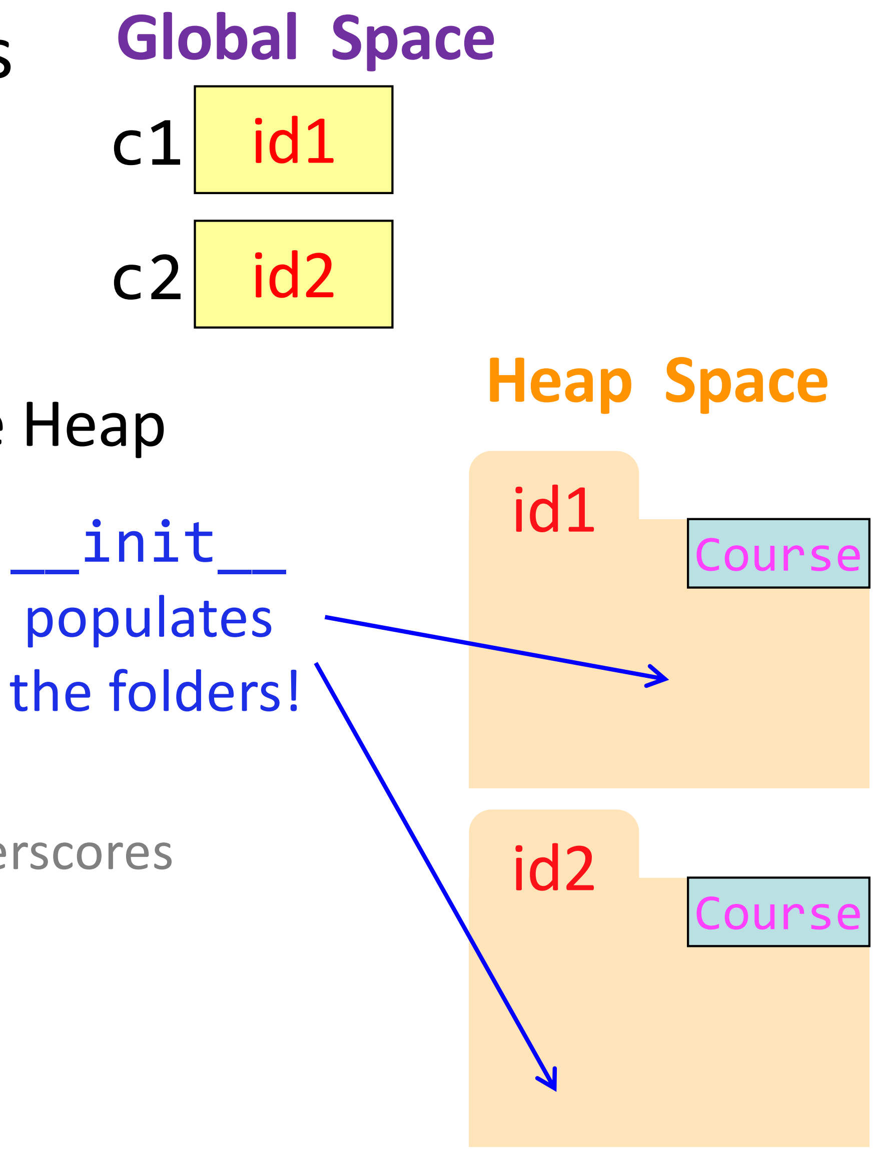


```
c1 = Course("CS 1110", 4)
c2 = Course("MATH 1920", 3)
```

Constructor (2)

- Function to create new instances
  - function name is the class name
- Calling the constructor:
  - Makes a new object (folder) on the Heap
  - Calls the `__init__` method
  - Returns the id of the folder

two underscores



Global Space

c1 id1

c2 id2

Heap Space

id1 Course

id2 Course

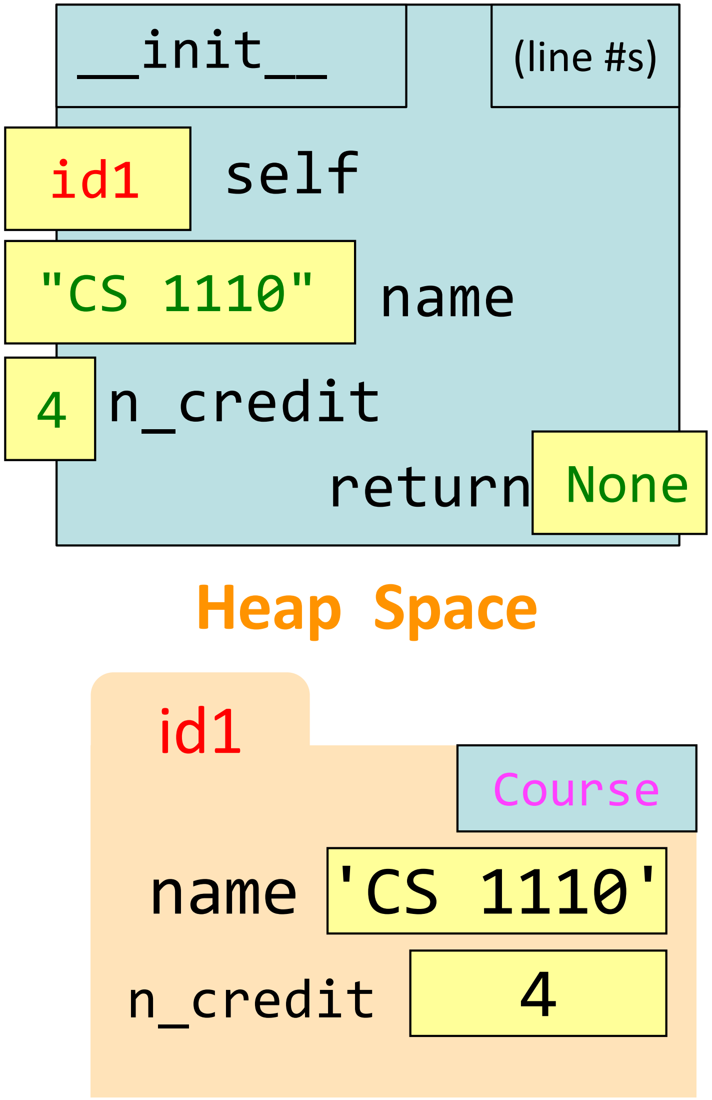
`__init__` populates the folders!

```
c1 = Course("CS 1110", 4)
c2 = Course("MATH 1920", 3)
```

Special Method: __init__

```
def  __init__(self, name, n_credit):
    """Initializer: creates a Course

    name:     [str] name of the course
    n_credit: [int] num credits, must be > 0
    """
    self.name  = name
    self.n_credit = n_credit
```

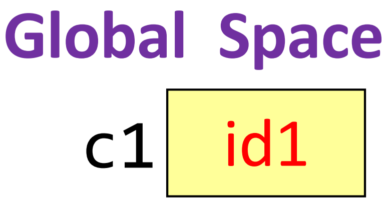
Param self: id of instance being initialized. Used to assign attributes

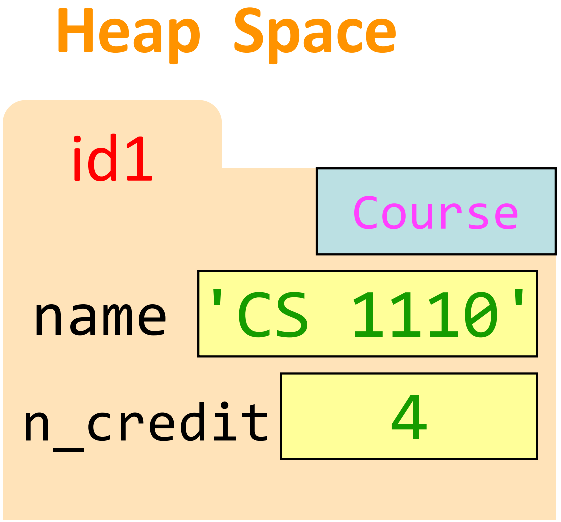


```
c1 = Course('CS 1110', 4)
# this is the call to the constructor, which calls __init__
```

Evaluating a Constructor Expression

1. Constructor creates a new object (folder) of the class `Course` on the Heap
   - Folder is initially empty
   - Has id
2. Constructor calls `__init__ (self, "CS 1110", 4)`
   - `self` = identifier (*"Fill this folder!"*)
   - Other args come from the constructor call
   - commands in `__init__` populate folder
   - `__init__` has no return value! (*"I filled it!"*)
3. Constructor returns the id
4. LHS variable created, id is value in the box





```
c1 = Course("CS 1110", 4)
```

Truths about Object Instantiation

1) Instantiate an object by calling the constructor
2) The constructor creates the folder
3) A constructor calls the `__init__` method
4) `__init__` puts attributes in the folder
5) The constructor returns the id of the folder

Invariants

- Properties of an attribute that must be true
- Works like a precondition:
  - If invariant satisfied, object works properly
  - If not satisfied, object is “corrupted”
- **Example**:
  - `Course` class: attribute `name` must be a string
- Purpose of the **class specification**

Checking Invariants with an Assert

```
class Course:
    """Instance is a Cornell course  """

    def __init__(self, name, n_credit):
    """Initializer: instance with name, n_credit courses
        name: [str] name of the course of form: <DEPT NUM>
        n_credit:  [int] num credits, must be > 0
    """
       assert type(name) == str, "name should be type str"
       assert name[0].isalpha(), " name should begin with a letter"
       assert name[-1].isdigit(), " name should end with an int"
       assert type(n_credit) == int, "n_credit should be type int"
       assert n_credit > 0, "n_credit should be > 0"

       self.name = name
       self.n_credit = n_credit
```

We know how to make:

- Class definitions
- Class specifications
- The `__init__` method
- Attributes (using `self`)

Let's make another class!

Student Class Specification, v1

```
class Student:
    """An instance is a Cornell student

    Instance Attributes:
    netID:      student netID [str], 2-3 letters + 1-4 digits
    courses:    list of courses
    major:      declared major [str]
    n_credit:   [int] num credits this semester
    """
```

Making Arguments Optional

- Can assign default values to `__init__` arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional

Examples:

```
s1 = Student(“xy1234”, [ ], "History")  # arguments 1,2,3

s2 = Student(“xy1234”, course_list)     # arguments 1 & 2

s3 = Student(“xy1234”, major="Art")     # arguments 1 & 3
```

```
class Student:
    def __init__(self, netID, courses=[ ], major=None):
        self.netID = netID
        self.courses = courses
        self.major = major
        # < the rest of initializer goes here >
```

default values when not specified

Student Class Specification, v2

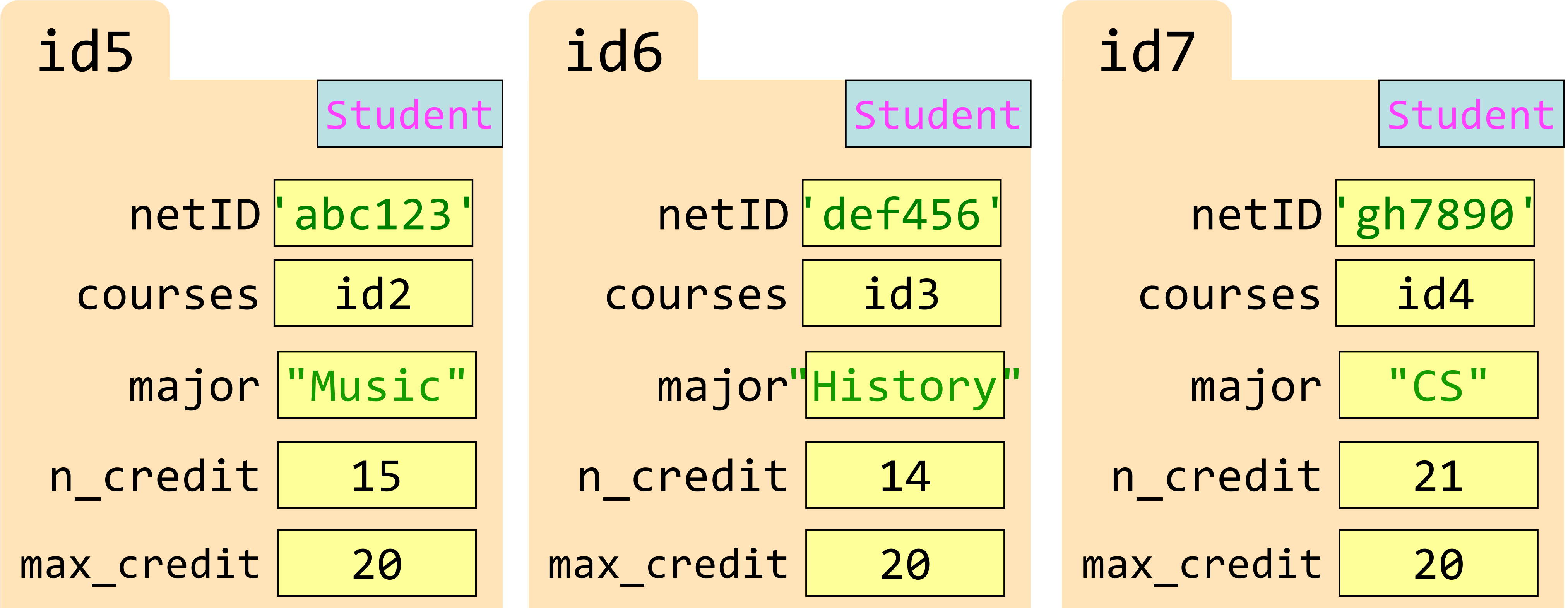
```
class Student:
    """An instance is a Cornell student

    Instance Attributes:
    netID:      student netID [str], 2-3 letters + 1-4 digits
    courses:    list of courses
    major:      declared major [str]
    n_credit:   [int] num credits this semester
    max_credit: [int] max num credits
    """
```

New attribute!

What do you think about this?

A look at three v2 Student instances

Anything wrong with this?



id5

Student

netID 'abc123'

courses id2

major "Music"

n_credit 15

max_credit 20

id6

Student

netID 'def456'

courses id3

major "History"

n_credit 14

max_credit 20

id7

Student

netID 'gh7890'

courses id4

major "CS"

n_credit 21

max_credit 20

Class Attributes

Class Attributes: Variables that belong to the Class

- One variable for the whole Class
- Shared by all object instances
- Access by `<Class Name>.<attribute-name>`

Why?

- Some variables are relevant to every object instance of a class
- Does not make sense to make them object attributes
- Doesn’t make sense to make them global variables, either

Example: we want all students to have the same credit limit

(Also in A4: `all_of_em` in both Song and `Mix`)

v3: Class Attributes – assign in class definition

```
class Student:
   """Instance is a Cornell student  """
   max_credit = 20
   def __init__(self, netID, courses, major):
      # < specs go here >
      < assertions go here >
      self.netID = netID
      self.courses = courses
      self.major = major
      self.n_credit = 0
      for c in courses: # add up all the credits
         self.n_credit = self.n_credit + c.n_credit
      assert self.n_credit <= Student.max_credit, "over credits!"
```

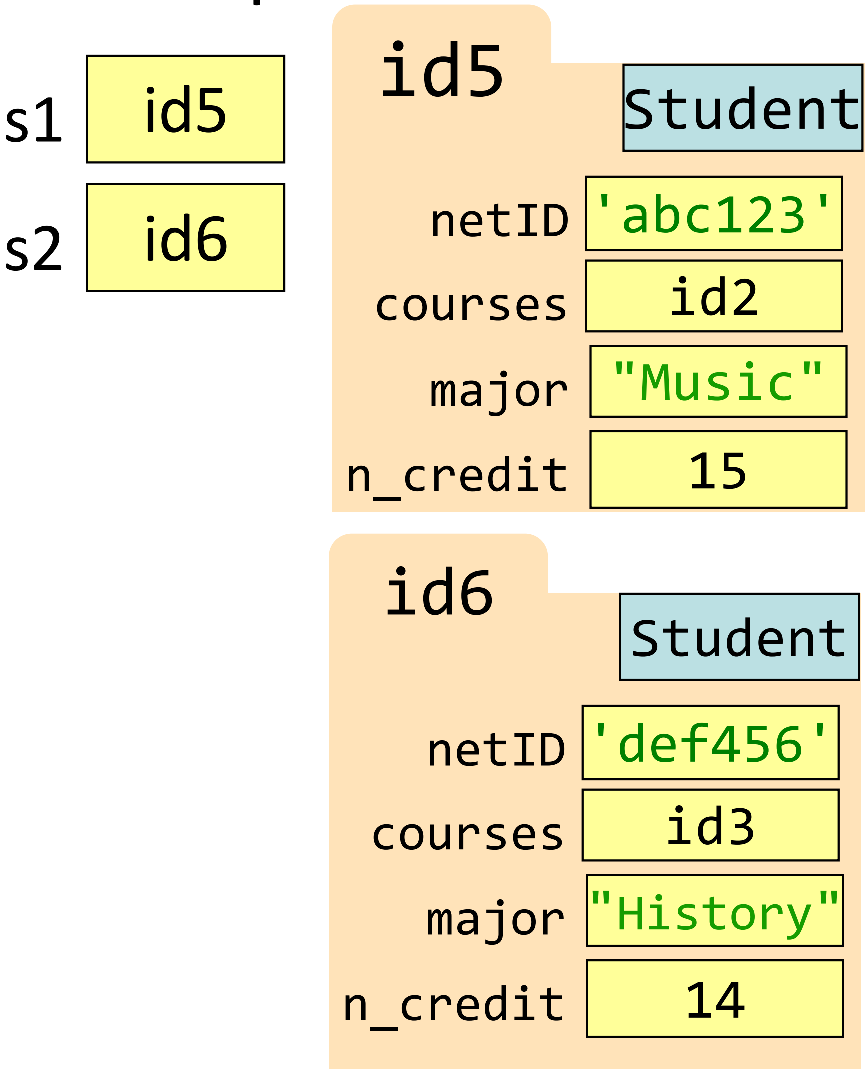
Where does max_credit live in memory?

Refer to class attribute using class name

Classes Have Folders Too

Object Folders

- Separate for each *instance*
- Example: *2* Student *objects*



s1 id5

s2 id6

id5 Student

netID 'abc123'

courses id2

major "Music"

n_credit 15

id6 Student

netID 'def456'

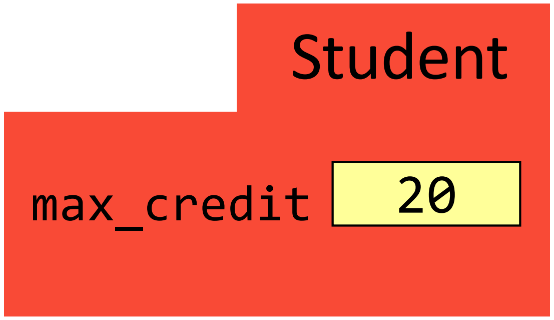
courses id3

major "History"

n_credit 14

Class Folders

- Data common to **all** instances



Student

max_credit 20

- Not just data!
- *Everything* common to all instances goes here!

Functions vs Object Methods

Function: call with object as argument

function name *function argument*

```
len(my_list)
print(my_list)
```

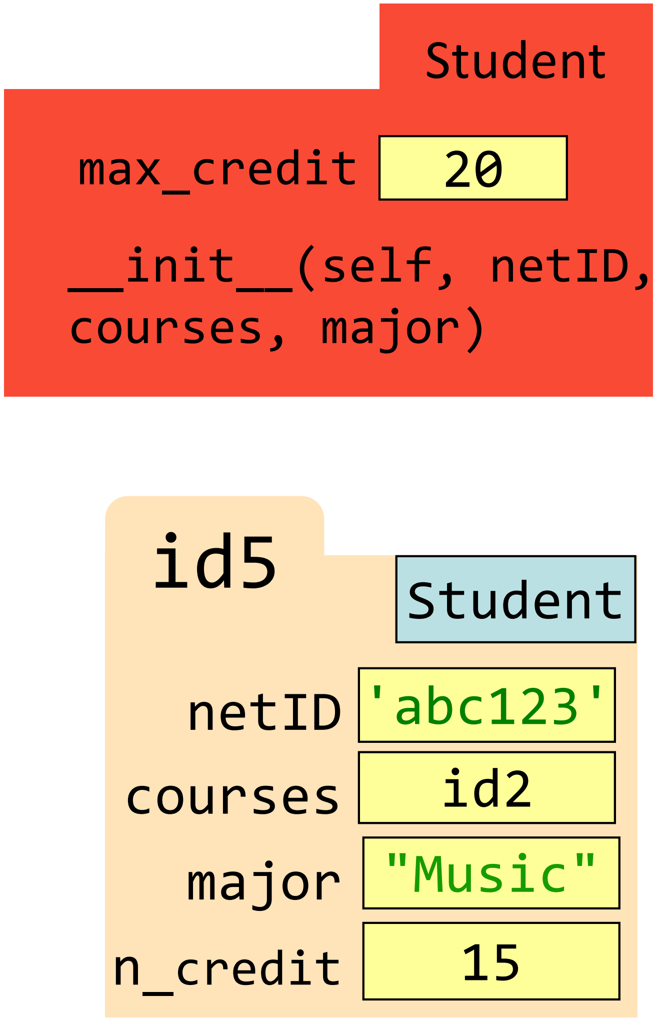
Method: function tied to the object

object variable *method name*

```
my_list.count(7)
my_list.sort()
```

Object Methods

- Attributes live in object folder
- Class Attributes live in class folder
- Methods live in class folder



Student

max_credit 20

```
__init__(self, netID, courses, major)
```

id5 Student

netID 'abc123'

courses id2

major "Music"

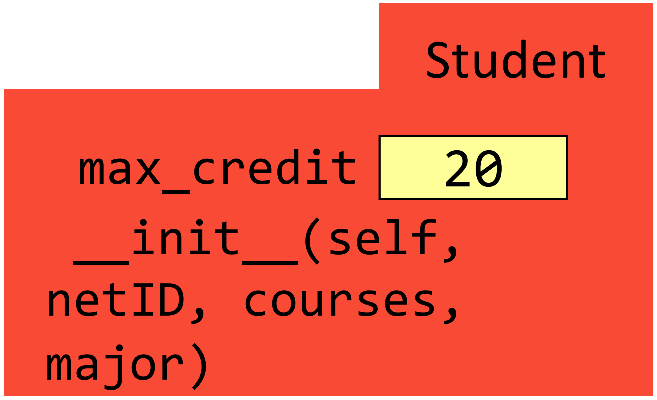
n_credit 15

Complete Class Definition

```
class <class-name>:
    """Class specification"""
    <assignment statements>
    <method definitions>
```

Look like function definitions:

- But indented *inside* class
- 1st parameter always `self`



Python creates the Class folder after reading the class definition

```
class Student():
    """Specification goes here.""”
    max_credit = 20
    def  __init__(self, netID, courses, major):
       . . . <snip>  . . .
```

Another Method Definition

```
c1 = Course("AEM 2400", 4)
s1.enroll(c1)
```

- `enroll` is defined in `Student` class folder
- `enroll` is called with `s1` as its first argument
- `enroll` knows which instance of `Student` it is working with

```
class Student():
    def __init__(self, netID, courses=[ ], major=None):
        # < init fn definition goes here >
    def enroll(self, new_course):
        if self.n_credit + new_course.n_credit > Student.max_credit:
            print("Sorry your schedule is full!")
        else:
            self.courses.append(new_course)
            self.n_credit = self.n_credit + new_course.n_credit
            print("Welcome to "+ new_course.name)
```

More Method Definitions!

```
class Student:
    def __init__(self, netID, courses=[], major=None):
        # < init fn definition goes here >
    def enroll(self, name, n):
        # < enroll fn definition goes here >
    def drop(self, course_name):
        """removes course with name course_name from courses list
           updates n_credit accordingly
           course_name: name of course to drop [str] """
        for one_course in self.courses:
           if one_course.name == course_name:
               self.n_credit = self.n_credit – one_course.n_credit
               self.courses.remove(one_course)
               print("just dropped "+course_name)
        print("currently at"+str(self.n_credit)+" credits")
```

We now know how to make:

- Class definitions
- Class specifications
- The `__init__` function
- Attributes (using `self`)
- Class attributes
- Class methods

Rules to live by (1/1)

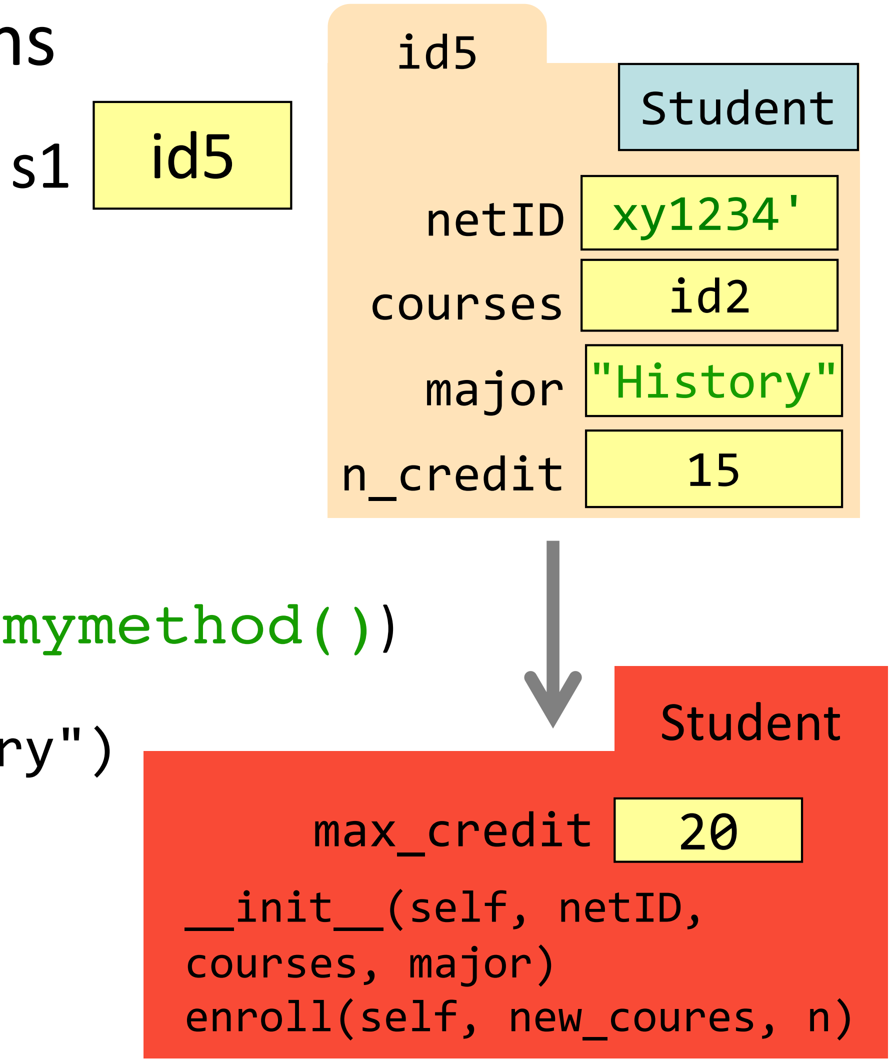
1. Refer to Class Attributes using the Class Name

```
s1 = Student("xy1234", [ ], "History")
print("max credits = " + str(Student.max_credit))
```

Name Resolution for Objects

- `myobject.myattribute` means
  - Go the folder for `myobject`
  - Find method `myattribute`
  - If missing, check class folder
  - If not in either, raise error

  (Same thing applies to `myobject.mymethod()`)



```
s1 = Student("xy1234", [ ], "History")
# finds attribute in object folder
print(s1.netID)
# finds attribute in class folder
print(s1.max_credit)
```
← dangerous

Accessing vs. *Modifying* Class Variables

- **Recall:** you cannot assign to a global variable from inside a function call
- **Similarly:** you cannot assign to a class attribute from “inside” an object variable

```
s1 = Student(“xy1234”, [ ], "History")
Student.max_credit = 23  # updates class attribute
s1.max_credit = 24       # creates new object attribute
                         #    called max_credit
```

Better to refer to Class Variables using the Class Name

Just like it did in the __init__ method!

What gets Printed? (Q)

```
import college

s1 = college.Student(“jl200", [], "Art")
print(s1.max_credit)
s2 = college.Student(“jl202", [], "History")
print(s2.max_credit)
s2.max_credit = 23
print(s1.max_credit)
print(s2.max_credit)
print(college.Student.max_credit)
```

A:
20
20
23
23
23

B:
20
20
23
23
20

C:
20
20
20
23
20

D:
20
20
20
23
23

What gets Printed? (A)

```
import college

s1 = college.Student(“jl200", [], "Art")
print(s1.max_credit)
s2 = college.Student(“jl202", [], "History")
print(s2.max_credit)
s2.max_credit = 23
print(s1.max_credit)
print(s2.max_credit)
print(college.Student.max_credit)
```

A:
20
20
23
23
23

B:
20
20
23
23
20

C:
20
20
20
23
20

D:
20
20
20
23
23

CORRECT

Rules to live by (2/2)

1. Refer to Class Attributes using the Class Name

```
s1 = Student("xy1234", [ ], "History")
print("max credits = " + str(Student.max_credit))
```

2. Don’t forget `self`
   - in parameter list of method (method header)
   - when defining method (method body)

Don’t forget self, Part 1

```
def enroll(~~self~~, new_course):  # if you forget self entirely
     if ~~self~~.n_credit + n > Student.max_credit:
           print("Sorry your schedule is full!")
     else:
           self.courses.append(new_course)
           ~~self~~.n_credit = ~~self~~.n_credit + new_course.n_credit
           print("Welcome to "+ new_course.name)
```

```
s1 = Student(“xy1234”, [ ], "History")
c5 = Course("AEM 2400", 4)
s1.enroll(c5)
```

always passes s1 as first argument!

```
TypeError: enroll() takes 1 positional
arguments but 2 were given
```

Don’t forget self, Part 2

```
def enroll(self, new_course):  # if you forget self in the body
     if ~~self~~.n_credit + n > Student.max_credit:
           print("Sorry your schedule is full!")
     else:
           self.courses.append(new_course)
           ~~self~~.n_credit = ~~self~~.n_credit + new_course.n_credit
           print("Welcome to "+ new_course.name)
```

```
s1 = Student(“xy1234”, [ ], "History")
c5 = Course("AEM 2400", 4)
s1.enroll(c5)
```

```
NameError: global name
          ‘n_credit' is not defined
```