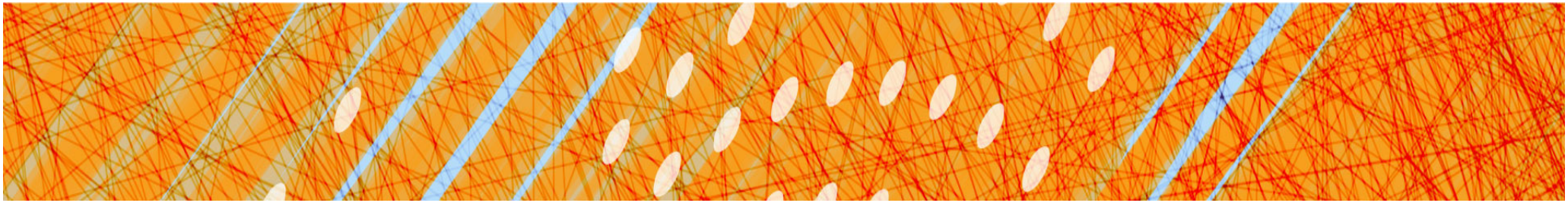


Lecture 15:
Classes
(Chapters 15 & 17.1-17.5)

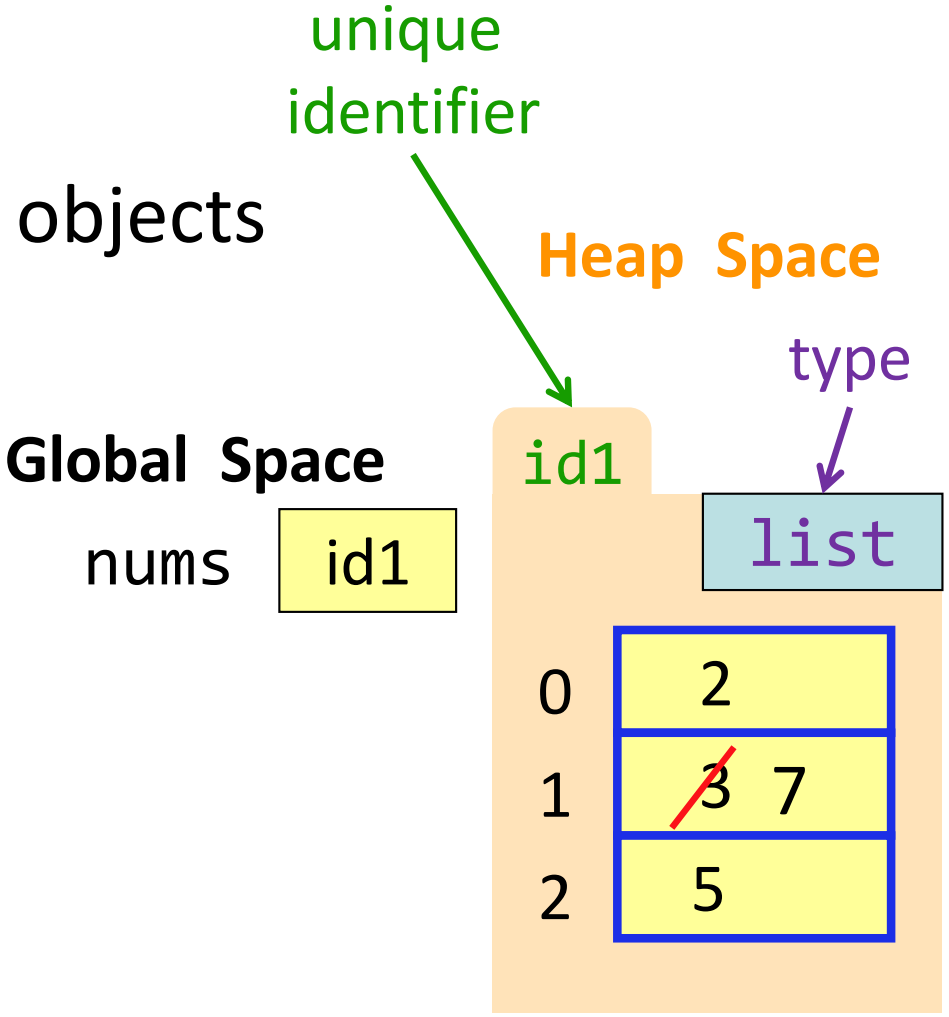
CS 1110

Introduction to Computing Using Python

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

Recall: Objects as Data in Folders

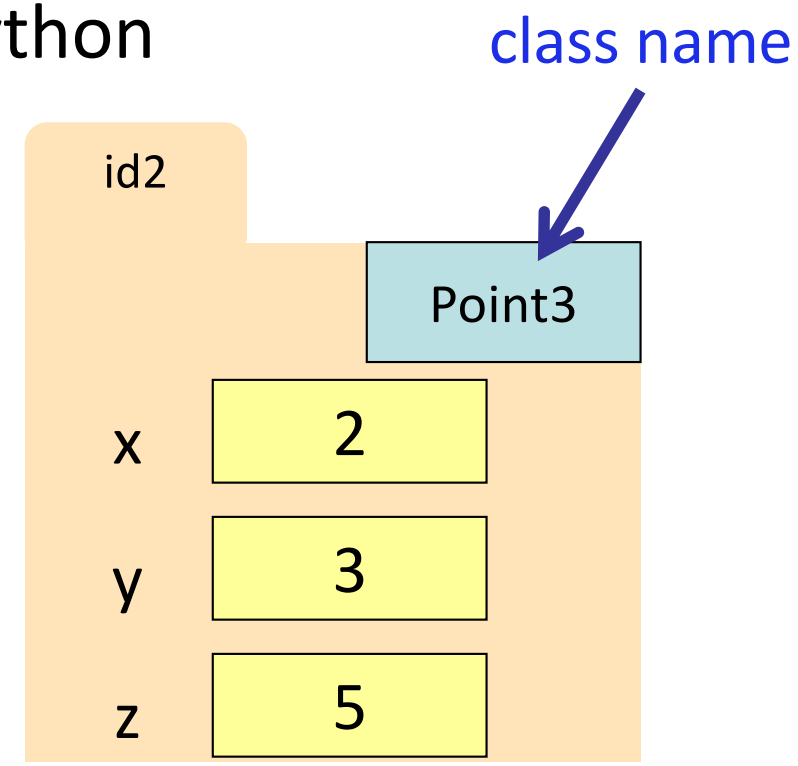
- attributes: variables within objects
- Type shown in the corner




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

Classes are user-defined Types

Defining new classes = adding new types to Python



Example Classes

- Point3
- Rect
- FreqText (A3), for letter frequency statistics
- Doll (class, lab)
- Student, Food (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

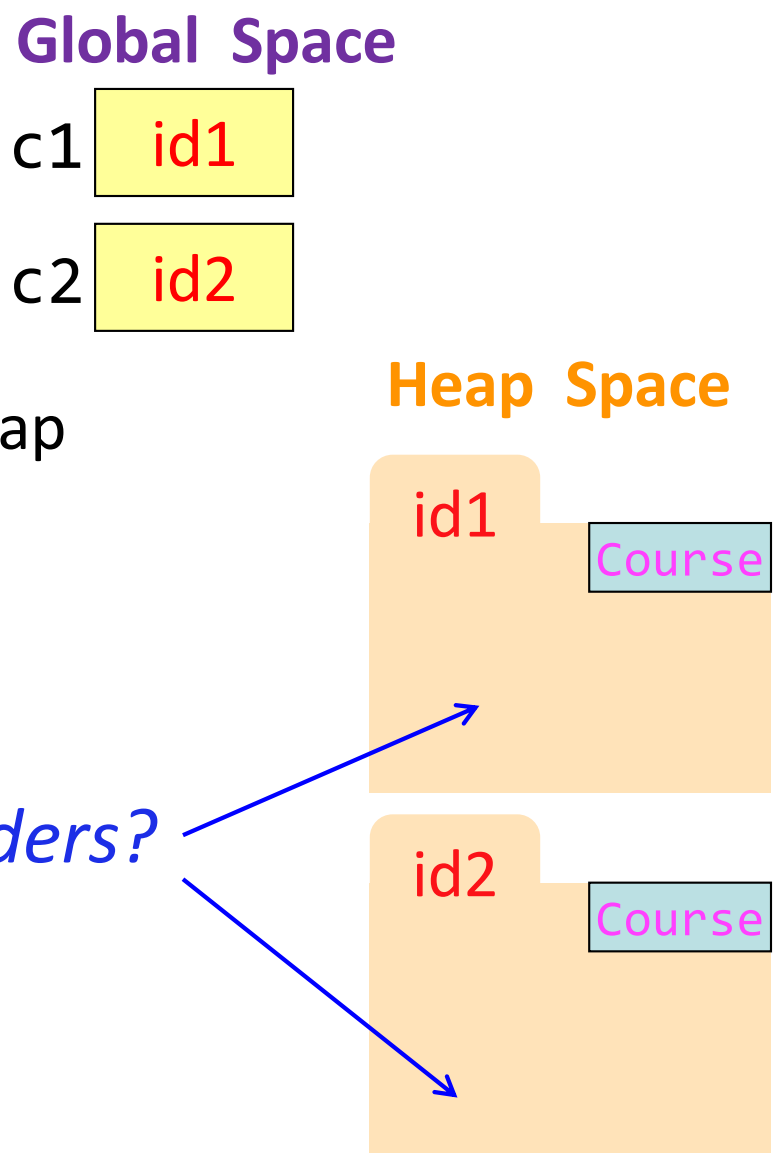
*Description and invariant**

**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

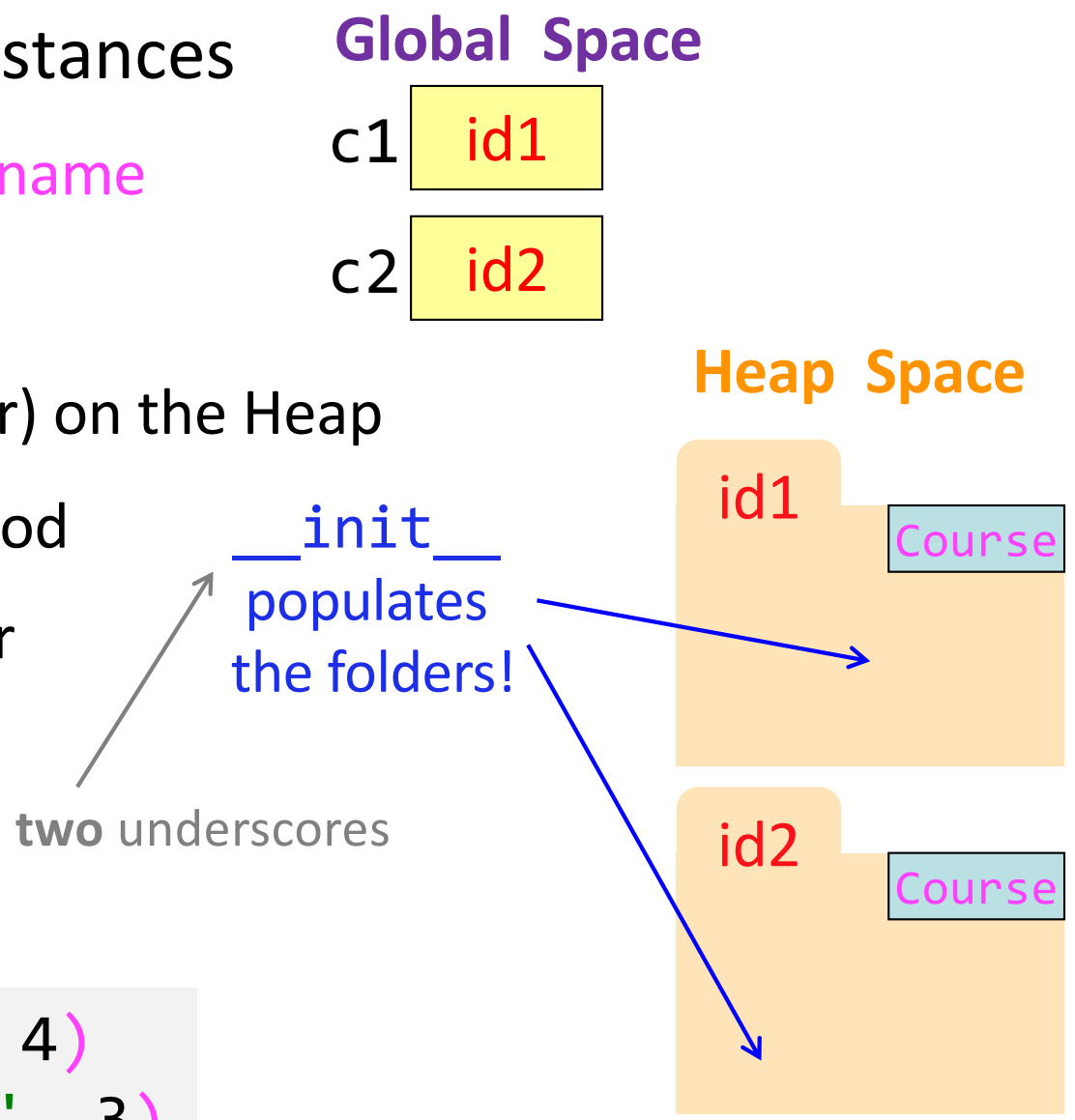


But how do we populate the folders?

```
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

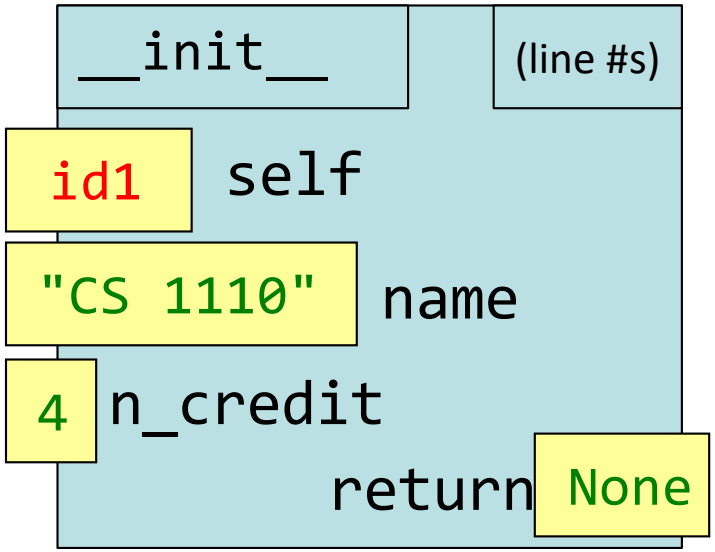



```
c1 = Course("CS 1110", 4)
c2 = Course("MATH 1920", 3)
```

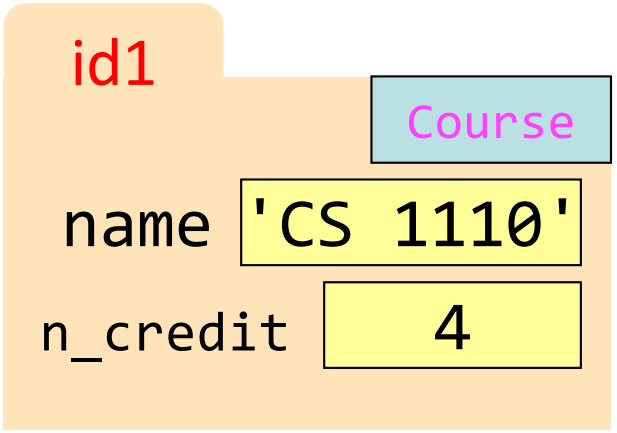
Special Method: __init__

```
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



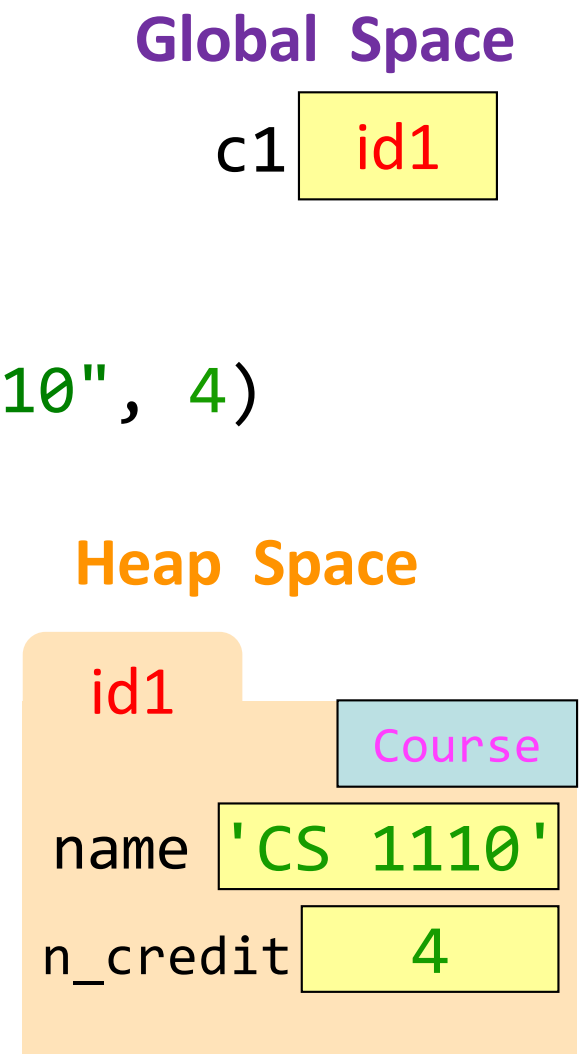
Heap Space




```
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



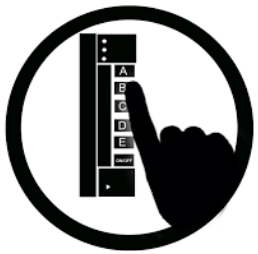

```
c1 = Course("CS 1110", 4)
```

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

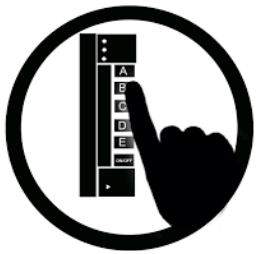
Which statement is true?

A) A constructor returns an id.
B) A constructor returns None.
C) A constructor returns True if it was successful and False if it failed to create the object.
D) I don't know.



Which statement is true?

A) An __init__ method returns an id.
B) An __init__ method returns None.
C) An __init__ method True if it was successful and False if it failed to create the object.
D) I don't know.



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:

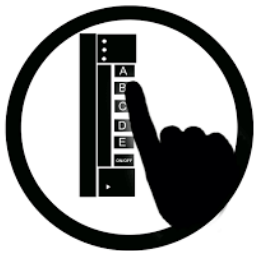
```
s1 = Student(“xy1234”, [ ], "History")  # arguments 1,2,3

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

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

```
class Student:
    def __init__(self, netID, courses= None, major=None):
        self.netID = netID
        self.courses = courses if courses is not None else []
        self.major = major
        # < the rest of initializer goes here >
```

default values when not specified

Which statement is true?

A) Python programmers implement constructors.
B) Python programmers design the way constructors can be called.
C) Python programmers implement __init__ methods.
D) B & C are both true
E) A B & C are all true

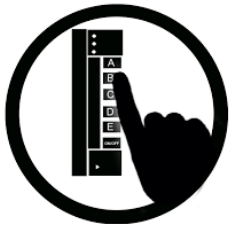


Student Class Specification, v2

```
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?



Take a look at 3 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

There is a problem with the field:

(A) netID (B) courses (C) major (D) n_credit (E) max_credit

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

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!"
```

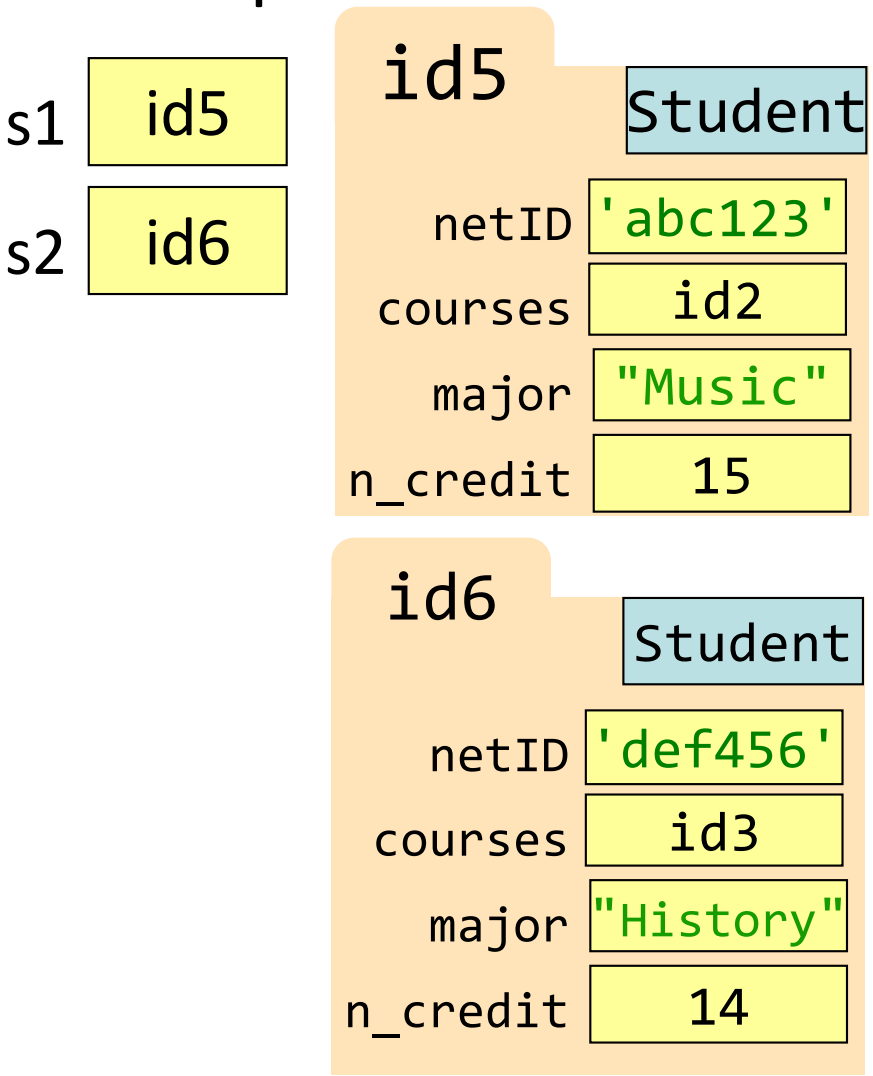
Where does max_credit live in memory?

Refer to class attribute using class name

Classes Have Folders Too

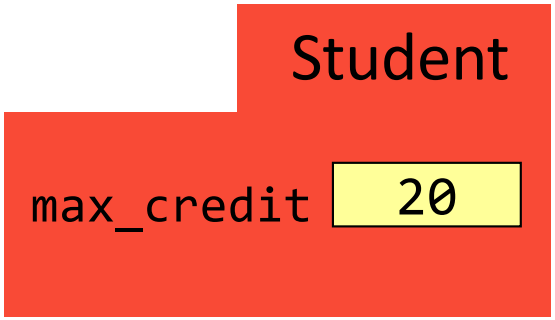
Object Folders

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



Class Folders

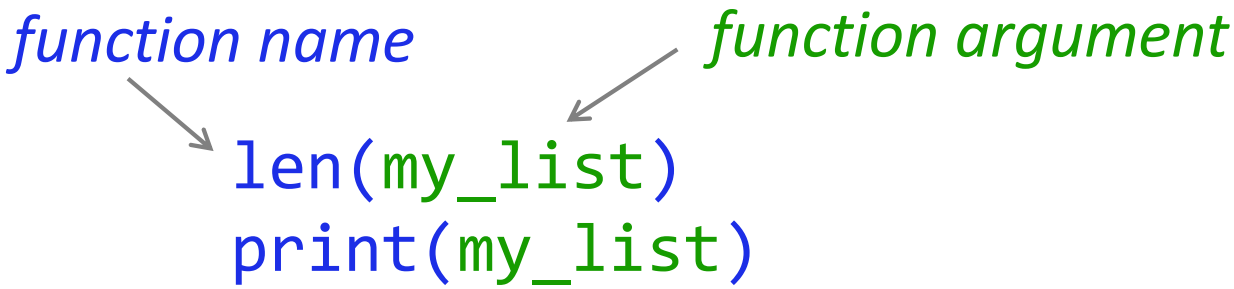
- Data common to **all** instances



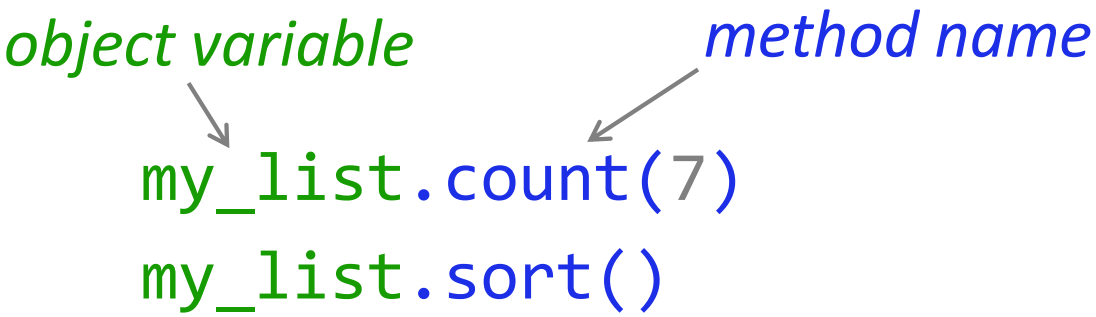
- Not just data!
- *Everything* common to all instances goes here!

Functions vs Object Methods

Function: call with object as argument

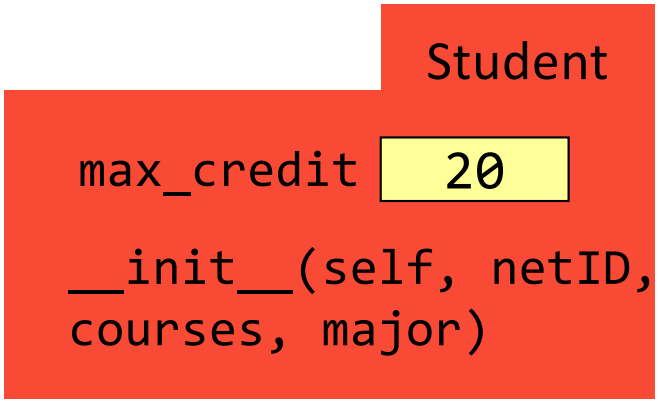


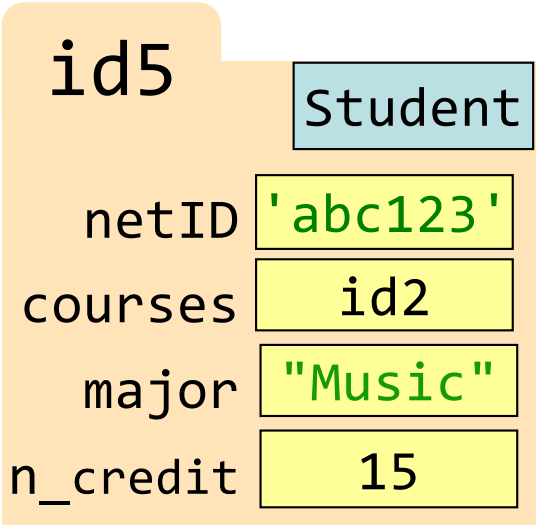
Method: function tied to the object



Object Methods

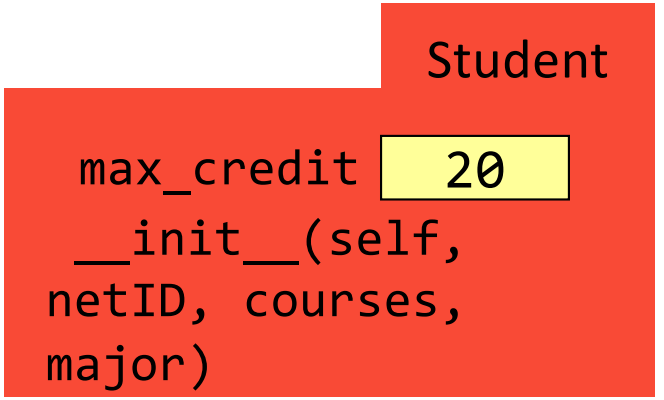
- Attributes live in object folder
- Class Attributes live in class folder
- Methods live in class folder





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=None, 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=None, 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")
```