http://www.cs.cornell.edu/courses/cs1110/2018sp

Lectures 17 & 18: Classes (Chapters 15 & 17)

CS 1110

Introduction to Computing Using Python

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

What got covered when?

Lecture 17

Slides 3-13, 16-19

Lecture 18

Slides 14, 15, 20-42

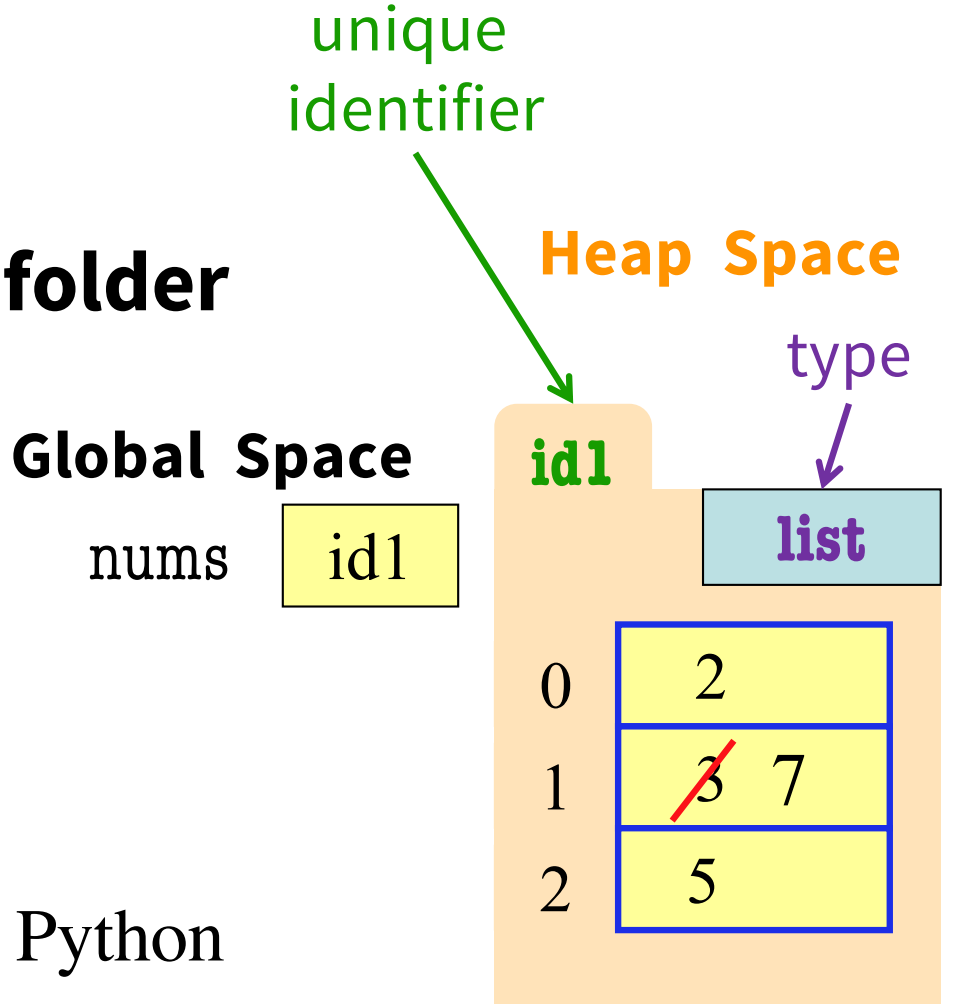
Appendix

Slides 43-46

Recall: Objects as Data in Folders

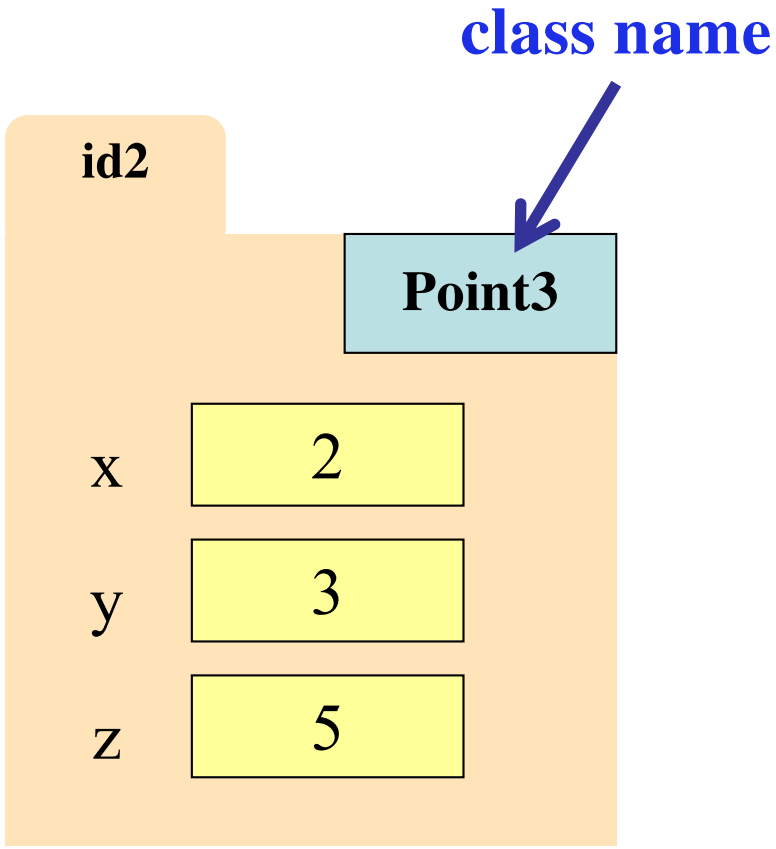
```
nums = [2,3,5]
nums[1] = 7
```

- An object is like a **manila folder**
- Contains variables
  - called **attributes**
  - Can change attribute values (w/ assignment statements)
- **Tab** identifies it
  - Unique number assigned by Python
  - Fixed for lifetime of the object
- **Type** listed in the corner



Classes are user-defined Types

Classes are how we add new types to Python



Example Classes

- Point3
- Card
- Rect
- Person

Simple Class Definition

class <*class-name*>():

    """Class specification"""

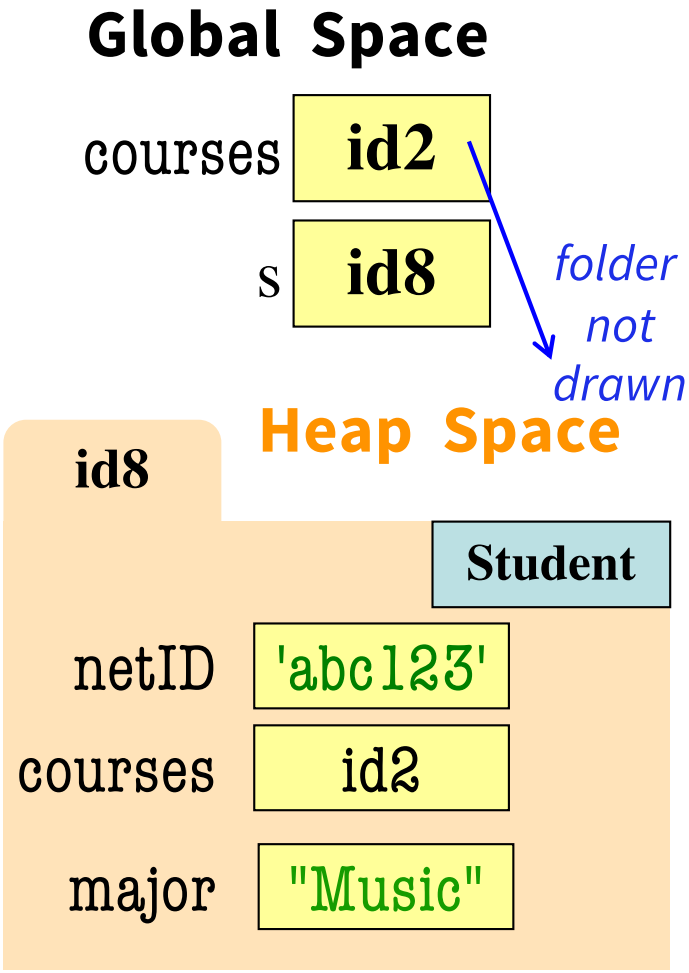
    <*method definitions*>

The Class Specification

```
class Student():
    """Instance is a Cornell student

    Instance Attributes:
    netID:      student's netID [str], 2-3 letters + 1-4 numbers
    courses:    list of tuples (name [str], n [int])
                name is course name, n is num credits
    major:      declared major [str]
    """
```

Constructors

- Function to create new instances
  - function name is the class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes (see next slide)
  - Returns the id of the folder




```
courses = [("CS 1110", 4), ("MATH 1920", 3)]
s = Student("abc123", courses, "Music")
```

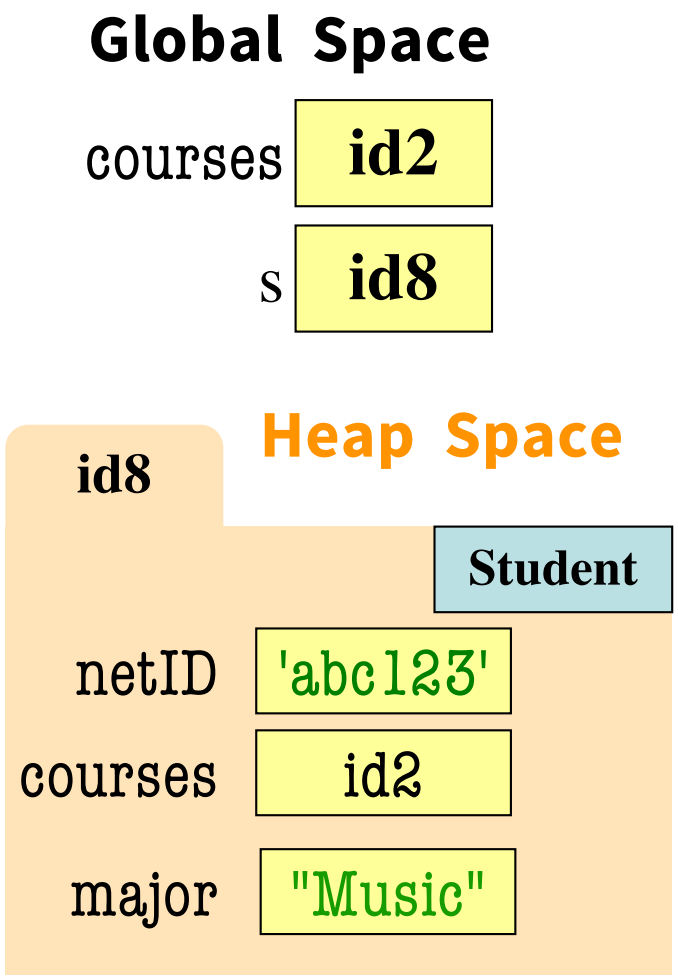
two underscores

Special Method: __init__

```
def __init__(self, netID, courses, major):
    """Initializer: creates a Student

    Has netID, courses and a major

    netID: [str], 2-3 letters + 1-4 numbers
    courses:   list of tuples (name [str], n [int])
               name is course name, n is number of credits
    major:   declared major [str]
    self.netID  = netID
    self.courses = courses
    self.major = major
```

called by the **constructor**

use self to assign attributes

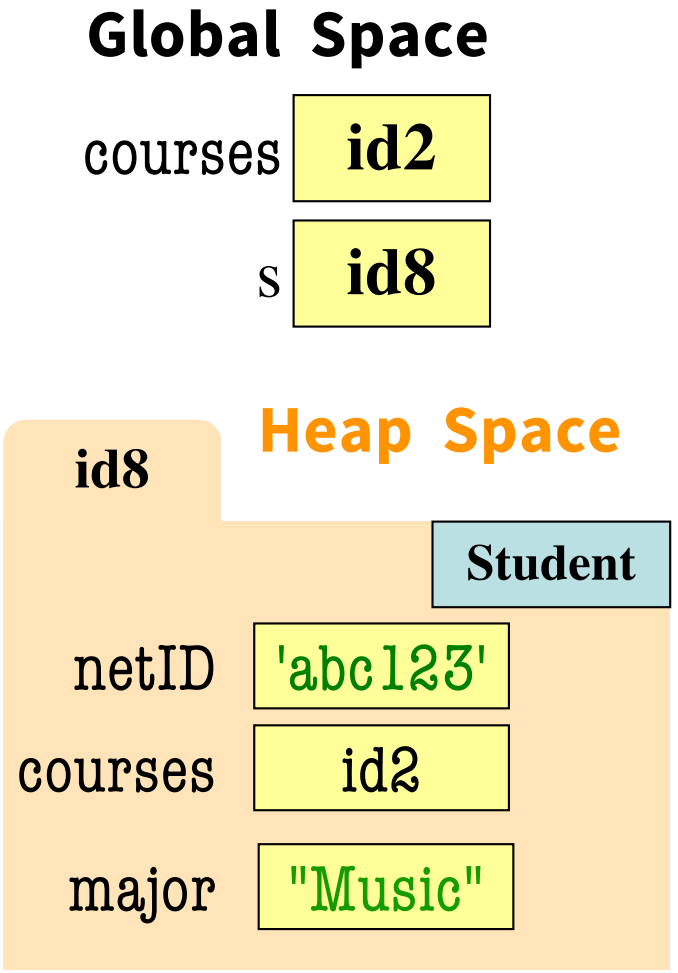



```
s = Student("abc123", courses, "Music")
# this is the call to the constructor, which calls __init__
```

Evaluating a Constructor Expression

```
s = Student("abc123", courses, "Music")
```

1. Creates a new object (folder) of the class Student on the heap
   - Folder is initially empty
2. Executes the method __init__
   - self = folder name = identifier
   - Other arguments passed in order
   - Executes commands in initializer
3. Returns folder name, the identifier



We know how to make:

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

Which statement is false?

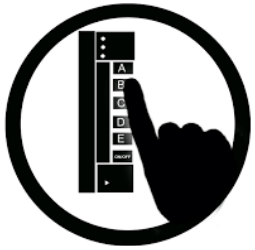
A) The constructor creates the folder

B) A constructor calls the __init__ method

C) The constructor returns the id of the folder

D) __init__ puts attributes in the folder

E) __init__ 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"
- **Examples**:
  - Point3 class: all attributes must be ints
  - RGB class: all attributes must be ints in 0..255
- Purpose of the **class specification**

  (see example on slide 5)

Checking Invariants with an Assert

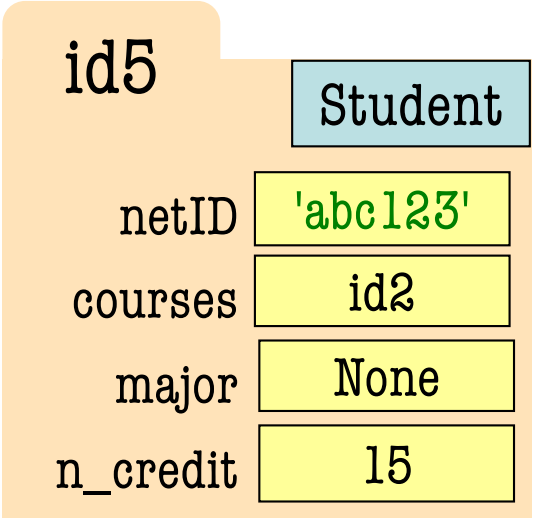
```
class Student():
    """Instance is a Cornell student  """
def __init__(self, netID, courses, major):
        """Initializer: instance with netID, and courses which defaults empty
        netID: [str], 2-3 letters + 1-4 numbers
        courses:   list of tuples (name [str], n [int])
                   name is course name, n is number of credits
        major:     declared major [str]   """
    assert type(netID) == str, "netID should be type str"
    assert netID[0].isalpha(), "netID should begin with a letter"
    assert netID[-1].isdigit(), "netID should end with an int"
    assert type(courses) == list, "courses should be a list"
    assert type(major) == str, "major should be type str"
    self.netID = netID
    self.courses = couress
    self.major = major
```

Aside: The Value `None`

- The `major` field is a problem.
  - `major` is a declared major
  - Some students don't have one!

Solution: use value `None`

- **`None`**: Lack of str
- Will reassign the field later!



Making Arguments Optional

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

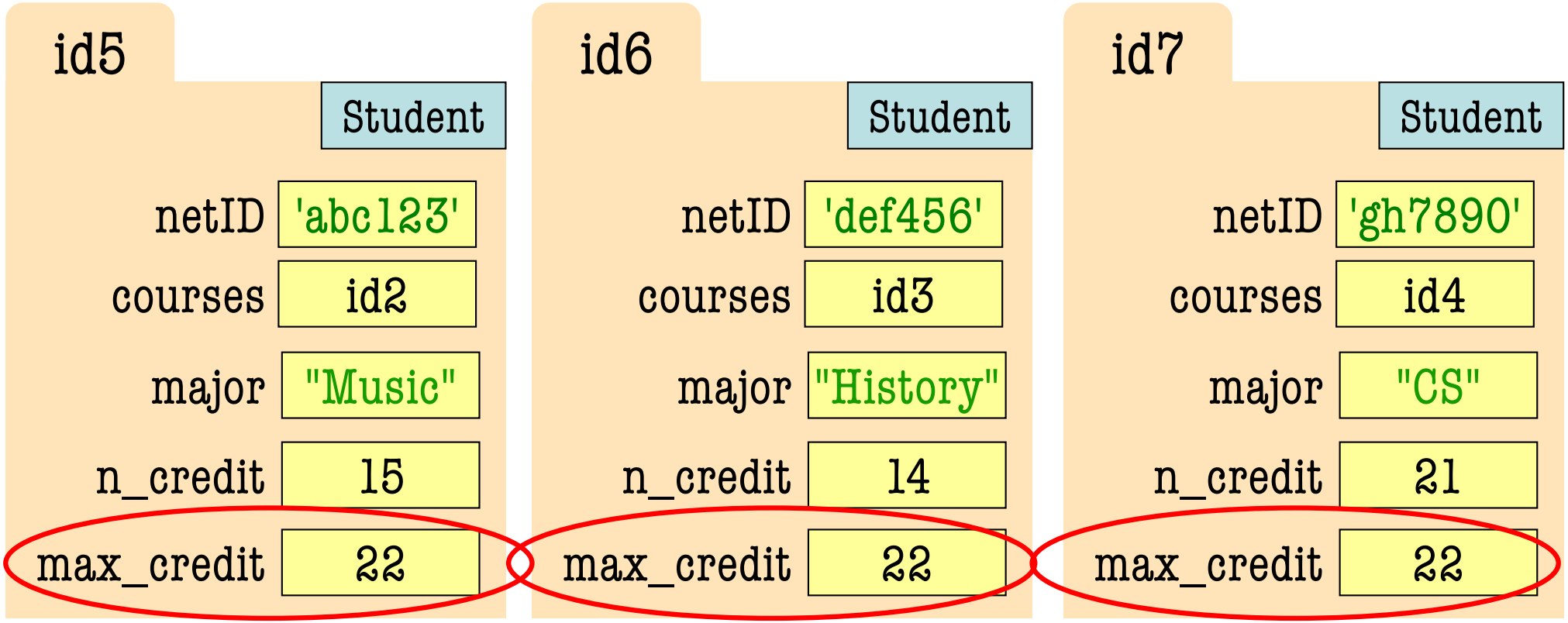
Examples:

```
s1 = Student("xy1234", [], "History")   # all parameters given

s1 = Student("xy1234", course_list)  # netID, courses given, major defaults to None

s1 = Student("xy1234", major="Art")  # netID, major given, courses defaults to []
```

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

What if…

We want to track and limit the number of credits a student is taking….



Anything wrong with this?

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

Class Attributes for CS1110

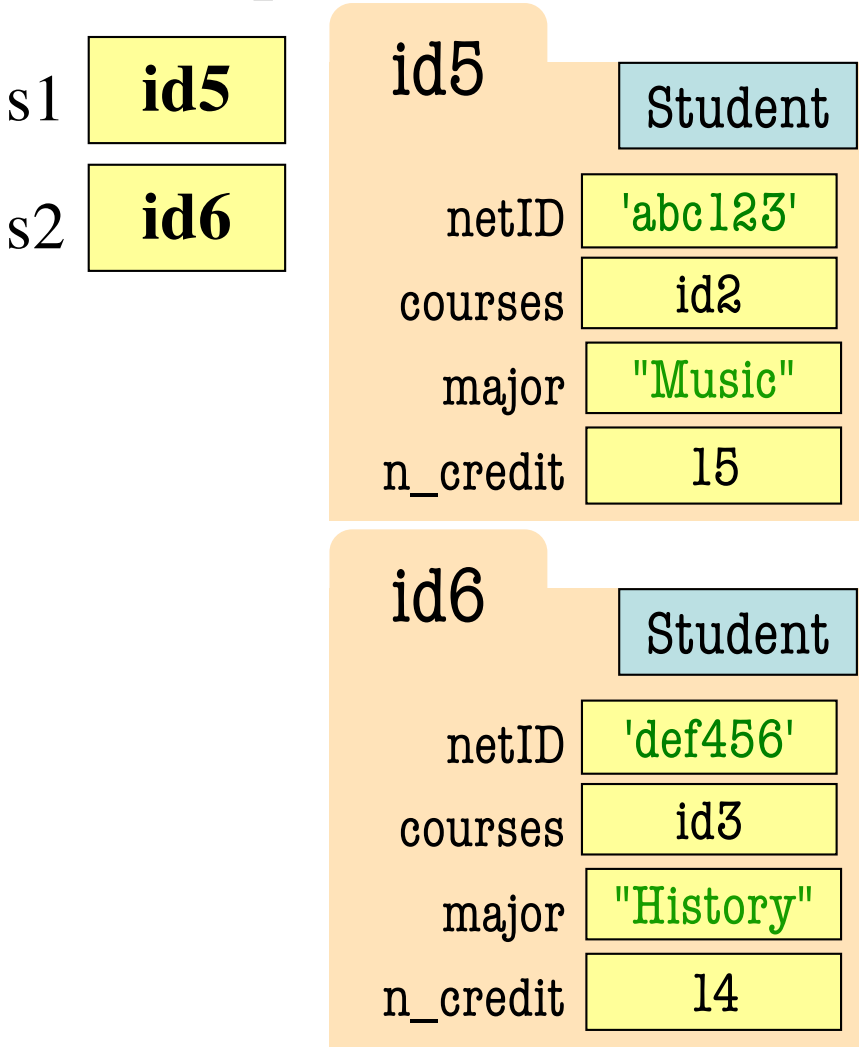
```
class Student():
    """Instance is a Cornell student  """
    max_credit = 22
     def __init__(self, NetID, courses, major):
         # < specs go here >
         # < assertions go here >
        self.netID = netID
        self.courses = couress
        self.major = major
        self.n_credit = 0
        for (course, n) in courses:
            self.n_credit = self.n_credit + n     # add up all the credits

         assert self.n_credit <= Student.max_credit, "over credit limit"
```

Where does **max_credit** live???

Classes Have Folders Too

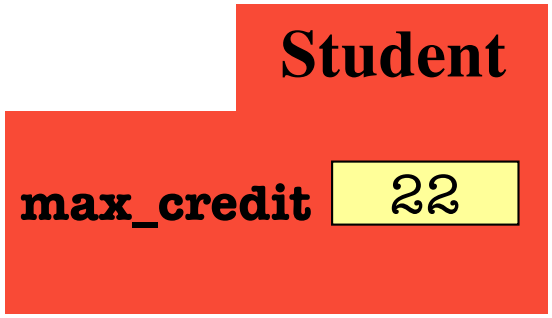
Object Folders

- 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!

Objects can have Methods

Function: call with object as argument

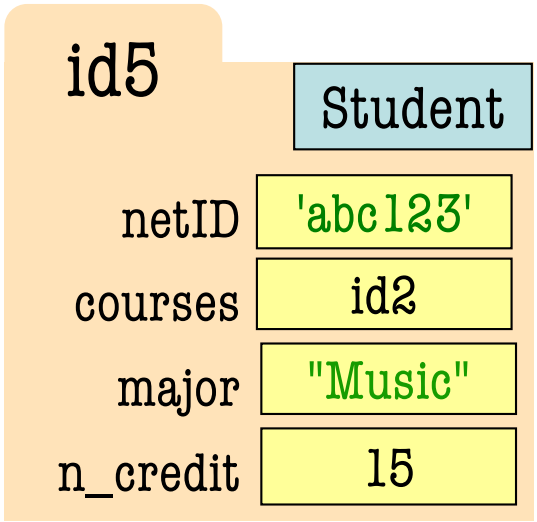
<function-name>(<arguments>)

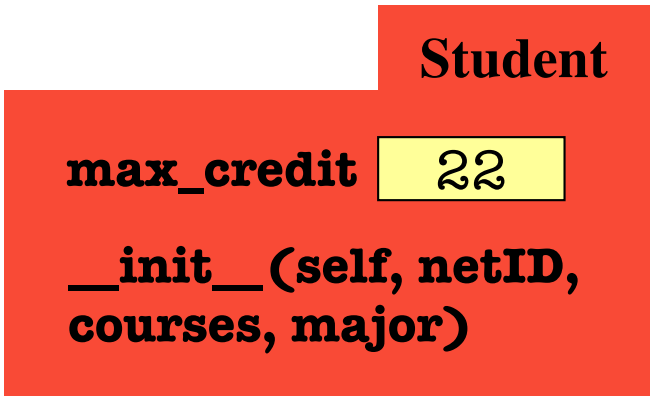
len(my_list)

Method: function tied to the object

<object-variable>.<function-call>

my_list.count(7)





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

Complete Class Definition

keyword class
Beginning of a class definition

class <*class-name*>():

Specification (similar to one for a function)

"""Class specification"""

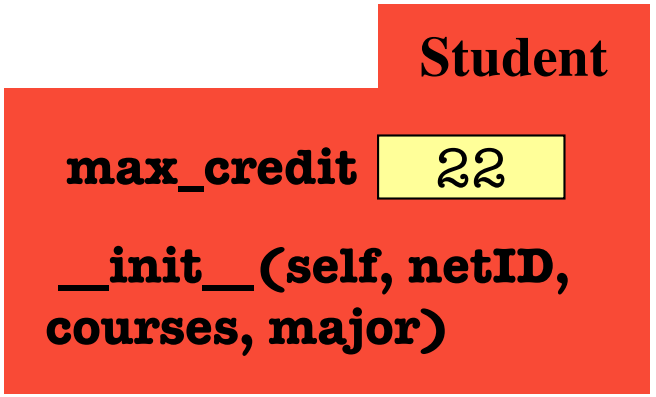
<*assignment statements*>

to define **class variables**

<*method definitions*>

to define **class methods**

```
class Student():
    """Specification goes here."""
    max_credit = 22
    def  __init__(self, netID, courses, major):
        . . . <snip>  . . .
```



Python creates after reading the class definition

Method Definitions

Looks like a function def

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

Example:

`s1.enroll("AEM 2400", 4)`

- Go to class folder for `s1` (*i.e.*, Student) that's where enroll is defined
- Now enroll is called with `s1` as its first argument
- Now enroll knows which instance of Student it is working with

Student

max_credit 22

__init__(self, netID, courses, major)

enroll(self, new_coures, n)

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

    def enroll(     self, name, n):
        if self.n_credit + n > Student.max_credit:
            print("Sorry your schedule is full!")
        else:
            self.courses.append((name, n))
            self.n_credit = self.n_credit + n
            print("Welcome to "+ 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 tuple with name new_course from courses list
        updates n_credit accordingly
        course_name: name of course to drop [str] """
        for name,n  in self.courses:
            if name == course_name:
                self.n_credit = self.n_credit - n
                self.courses.remove((name,n))
                print("just dropped "+name)
                print("currently have "+str(self.n_credit)+" credits")
```

Data Encapsulation

- **Idea**: Force the user to only use methods
- Do not allow direct access of attributes

Setter Method

- Used to change an attribute
- Replaces all assignment statements to the attribute
- **Bad**:

```
>>> sl.major = "Anthropology"
```

- **Good**:

```
>>> sl.setMajor("Anthropology")
```

Getter Method

- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**:

```
>>> print("major: "+ sl.major)
```

- **Good**:

```
>>> print("major: "+ sl.getMajor())
```

Data Encapsulation

```
class Student():
    def __init__(self, NetID, courses, major):
        # < specs go here >
        # < assertion & definition goes here >
        self._major = major

    def getMajor(self):
        """Returns: major attribute"""
        if self._major == None:
            return ""
        return self._major

    def setMajor(self, m):
        """ Sets major to m
        Pre: m must be a major at Cornell """
        # could check major requirements
        self._major = m
```

Getter

Setter

Naming Convention

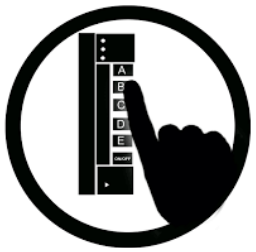
The underscore means "should not access the attribute directly."

Precondition is same as attribute invariant.

Should this be allowed?

```
courses = [("MATH 1920", 3), ("HADM 2200", 3), ("CS 1110", 4)]
s1 = Student("mep1", courses, "Economics")
s1.n_credit = 10    ← A
s1.n_creidt = 30    ← B
```

A) A should be allowed, but not B
B) B should be allowed, but not A
C) Both should be allowed
D) Neither should be allowed
E) I don't know



Hiding Methods From Access

- Put underscore in front of a method will make it **hidden**
  - Will not show up in help()
  - But it is still there…
- Hidden methods
  - Can be used as **helpers** of the same class
  - But it is bad style to use them outside of this class

HIDDEN

Helper method

```
class Student():
    max_credit = 22
    def __init__(self, NetID, courses, major):
        # < specs go here >
        # < assertions go here >
        # < definition goes here >
        self._major = major

    def _isMajor(m):  **
        """True if m is a major at Cornell"""
        return m == "Computer Science"

    def setMajor(self, m):
        """ Sets major to m
        Pre: m must be a major at Cornell """
        assert(Student._isMajor(m))
        self._major = m
```

** *Pretend CS is the only major at Cornell*

We know how to make:

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

Class Gotchas… and how to avoid them

Rules to live by:

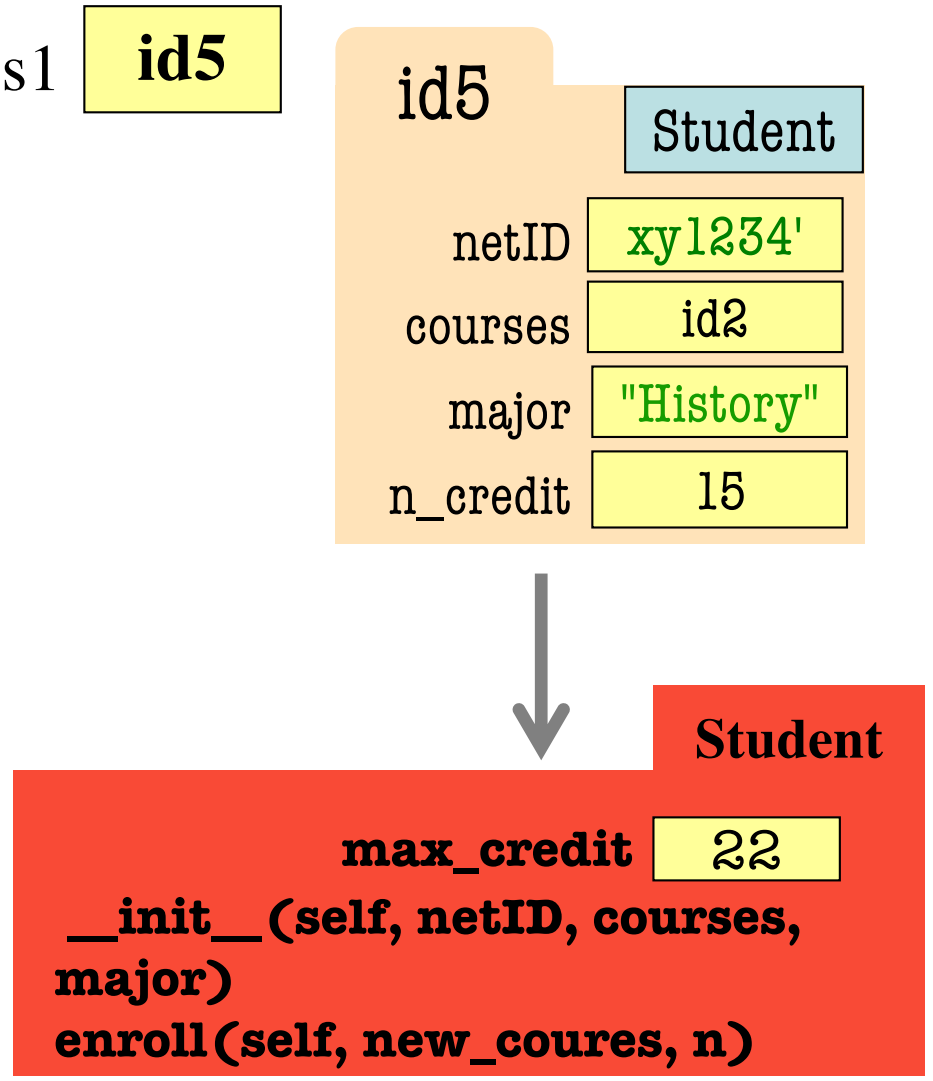
1. Refer to Class Attributes using the Class Name

```
s1 = Student(“xy1234”, [], "History")

print(“max credits = “+str(Student.max_credit))
```

Name Resolution for Objects

- ⟨*object*⟩.⟨*name*⟩ means
  - Go the folder for *object*
  - Find attribute/method *name*
  - If missing, check **class folder**
  - If not in either, raise error

```
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

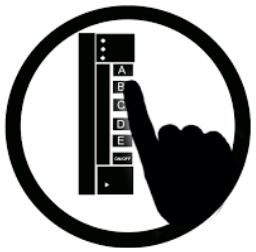
- **Recall:** you cannot assign to a global variable from inside a function call
- **Similarly:** you cannot assign to a class attribute from "inside" an object variable

```
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

What gets Printed? (Q)

```
import cs1110

s1 = cs1110.Student("jl200", [], "Art")
print(s1.max_credit)
s1 = cs1110.Student("jl202", [], "History")
print(s2.max_credit)
s2.max_credit = 23
print(s1.max_credit)
print(s2.max_credit)
print(cs1110.Student.max_credit)
```

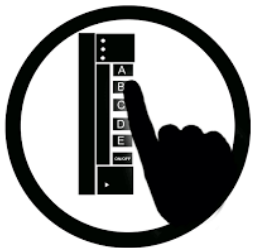


A:	B:
22	22
22	22
23	23
23	23
23	22

C:	D:
22	22
22	22
22	22
23	23
22	23

What gets Printed? (A)

```
import cs1110

s1 = cs1110.Student("jl200", [], "Art")
print(s1.max_credit)
s1 = cs1110.Student("jl202", [], "History")
print(s2.max_credit)
s2.max_credit = 23
print(s1.max_credit)
print(s2.max_credit)
print(cs1110.Student.max_credit)
```



A:	B:
22	22
22	22
23	23
23	23
23	22

C:	D:
22	22
22	22
22	22
23	23
22	23

CORRECT

Class Gotchas… and how to avoid them

Rules to live by:

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

Don't forget self, Part 1

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

<var>.<method_name> **always** passes <var> as first argument

```
class Student():
    def __init__(self, netID, courses, major):
        self.netID = netID
        self.courses = courses
        self.major = major
        # < rest of constructor goes here >

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

TypeError: enroll() takes 2 positional arguments but 3 were given

Don't forget self, Part 2 (Q)

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

What happens?

A) Error

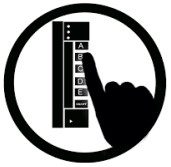
B) Nothing, self is not needed

C) creates new local variable n_credit

D) creates new instance variable n_credit

E) creates new Class attribute n_credit

 # if you forget self



```
class Student():
    def __init__(self, netID, courses, major):
        self.netID = netID
        self.courses = courses
        self.major = major
        # < rest of constructor goes here >

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

Don't forget self, Part 2 (A)

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

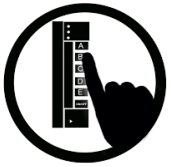
What happens?

A) Error
B) Nothing, self is not needed
C) creates new local variable n_credit
D) creates new instance variable n_credit
E) creates new Class attribute n_credit

```
class Student():
    def __init__(self, netID, courses, major):
        self.netID = netID
        self.courses = courses
        self.major = major
        # < rest of constructor goes here >

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

 # if you forget self

NameError: global name 'n_credit' is not defined



init is just one of many **Special Methods**

Start/end with 2 underscores

- This is standard in Python
- Used in all special methods
- **Also for special attributes**

__init__ for initializer

__str__ for str()

__eq__ for ==

__lt__ for <, ...

For a complete list, see

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

```
class Point2():
    """Instances are points in 2D space"""
    ...

    def __init__(self,x=0,y=0):
        """Initializer: makes new Point2"""
        ...

    def __str__(self):
        """Returns: string with contents"""
        return '('+str(self.x) + ',' + str(self.y) + ')'

    def __eq__(self, other):
        """Returns: True if both coordinates equal"""
        return self.x == other.x and self.y == other.y
```

We know how to make:

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

Designing Types

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, /, …)
  - Point3 (**set**: x,y,z coordinates; **ops**: distanceTo, …)
  - Card (**set**: suit * rank combinations; **ops**: ==, !=, <)
  - New ones to think about: Person, Worker, Image, Date, *etc*.

- To define a class, think of a ***type*** you want to make

Making a Class into a Type

1. What values do you want in the set?
   - What are the attributes? What values can they have?
   - Are these attributes shared between instances (class attributes) or different for each attribute (instance attributes)?
   - What are the *class invariants:* things you promise to keep true **after every method call** *(see* `n_credit` *invariant)*
2. What operations do you want?
   - This often influences the previous question
   - What are the *method specifications:* states what the method does & what it expects (preconditions)
   - Are there any special methods that you will need to provide?

Write your code to make it so!

A word about invariants & preconditions

- When implementing methods:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Appendix

Sample Classes for you to look over:

- Time
- Rectangle
- Hand (in poker)

Planning out a Class

```
class Time(object):
    """Class to represent times of day.
    INSTANCE ATTRIBUTES:
        hour: hour of day [int in 0..23]
        min:  minute of hour [int in 0..59]"""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""

    def isPM(self):
        """Returns: this time is noon or later."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Time instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Rectangle(object):
    """Class to represent rectangular region
    INSTANCE ATTRIBUTES:
        t:  y coordinate of top edge      [float]
        l:  x coordinate of left edge     [float]
        b: y coordinate of bottom edge [float]
        r: x coordinate of right edge     [float]
    For all Rectangles, l <= r and b <= t."""

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b]
        Pre: args are floats; l <= r; b <= t"""

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
           intersection of self with other."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Hand(object):
    """Instances represent a hand in cards.
    INSTANCE ATTRIBUTES:
        cards: cards in the hand [list of card]
    This list is sorted according to the
    ordering defined by the Card class."""

    def __init__(self, deck, n):
        """Draw a hand of n cards.
        Pre: deck is a list of >= n cards"""


    def isFullHouse(self):
        """Return: True if this hand is a full
        house; False otherwise"""

    def discard(self, k):
        """Discard the k-th card."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.