Lecture 17: Classes
(Chapters 15 & 17.1-17.5)

CS 1110
Introduction to Computing Using Python

 Corrections made after lecture are shown in orange

[E. Andersen, A. Bracy, D. Fan, D. Gries, L. Lee,
 S. Marschner, C. Van Loan, W. White]
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
- Timer
- Rect
- Person
Simple Class Definition

class <class-name>():

    """Class specification"""

    <method definitions>
class Student():

    """An instance is a Cornell student

    Instance Attributes:
    netID:        student's netID [str], 2-3 letters + 1-4 digits
courses:      nested list [ [name0, n0], [name1, n1], ... ]
                name is course name [str], n is number of credits [int]
    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

```python
courses = [ ['CS 1110', 4], ['MATH 1920', 3] ]
s = Student('abc123', courses, 'Music')
```
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 digits
courses: nested list [[name0, n0], [name1, n1], ...]
    name is course name [str],
    n is number of credits [int]
major: declared major [str]
    self.netID = netID
    self.courses = courses
    self.major = major"

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

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

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

A) Instantiate an object by calling the constructor
B) The constructor creates the folder
C) The constructor returns the id of the folder
D) A constructor calls the `__init__` method
E) `__init__` puts attributes in 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:
  ▪ Point3 class: all attributes must be ints
• Purpose of the class specification
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 digits
courses: nested list [ [name0, n0], [name1, n1], ... ]
    name is course name [str], n is number of credits [int]
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 major==None or type(major) == str, "major should be None or type str"

self.netID = netID
self.courses = courses
self.major = major
• The major attribute 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 []

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

- Class definitions
- Class specifications
- The `__init__` method
- Attributes (using `self`)
Start next video:
Class attributes and method definitions
We know how to make:

- Simple class definitions
- Class specifications
- The `__init__` method
- Attributes (using `self`)
Continue developing our class Student ...

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 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 = courses
        self.major = major
        self.n_credit = 0
        for one_course in courses:
            self.n_credit = self.n_credit + one_course[1]  # add up all the credits

        assert self.n_credit <= Student.max_credit, "over credit limit"
Classes Have Folders Too

Object Folders

- Separate for each instance
- Example: 2 Student objects

<table>
<thead>
<tr>
<th>Student id</th>
<th>Id</th>
<th>Name</th>
<th>Courses</th>
<th>Major</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>id5</td>
<td></td>
<td>id2</td>
<td>Music</td>
<td>15</td>
</tr>
<tr>
<td>s2</td>
<td>id6</td>
<td></td>
<td>id3</td>
<td>History</td>
<td>14</td>
</tr>
</tbody>
</table>

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

```python
<function-name>(<arguments>)
len(my_list)
```

**Method:** function tied to the object

```python
<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>():

"""Class specification"""

<assignment statements>

<method definitions>

class Student():

"""Specification goes here."""

    max_credit = 22
    def __init__(self, netID, courses, major):
        ... <snip> ...

Student

max_credit 22

__init__(self, netID, courses, major)

Python creates after reading the class definition
Method Definitions

Looks like a function def
  ▪ But indented inside class
  ▪ 1st parameter always self

Example:

```python
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

```python
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)
```

Student

max_credit 22
We now 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

   ```python
   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

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

```python
s1 = Student('xy1234', [], 'History')
print(s1.netID)  # finds attribute in object 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

```python
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*
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)
Class Gotchas… and how to avoid them

Rules to live by:

1. Refer to Class Attributes using the Class Name
   ```python
   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)
s1 = Student("xy1234", [], "History")
s2 = Student("ab132", [], "Math")
s1.enroll("AEM 2400", 4)

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

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)
s1 = Student("xy1234", [], "History")
s2 = Student("ab132", [], "Math")
s1.enroll("AEM 2400", 4)

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)

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