Recall: Objects as Data in Folders

- 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 shown in the corner

Simple Class Definition

```python
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

```python
class Student:
    ***An instance is a Cornell student
    Attribute list
    Description and invariant

    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]
```

Conventions: capitalize first letter of class name
Constructor

- 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")
```

What happens when constructor is called?

```
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__
  - if __init__ exists
  - Puts attributes in the folder
  - Note: constructor calls __init__ automatically if it exists
- Returns folder name, the identifier

Special Method: __init__

```
def __init__(self, netID, courses, major):
    #Initializer: creates a Student
    self.netID = netID
    self.courses = courses
    self.major = major
```

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__
  - if __init__ exists
  - Puts attributes in the folder
  - Note: constructor calls __init__ automatically if it exists
- 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) A constructor calls the __init__ method
D) __init__ puts attributes in the folder
E) 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:
  * Student class: attribute courses must be a list
- Purpose of the class specification
Checking Invariants with an Assert

```python
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]"
        self.netID = netID
        self.courses = courses
        self.major = major
        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"
```

Aside: The Value None

- 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:
```python
s1 = Student("xy1234", [], "History") # all 3 arguments given
s1 = Student("xy1234", course_list) # netID, courses given, major defaults to None
s1 = Student("xy1234", major="Art") # netID, major given, courses defaults to []
```

Continue developing our class Student ...

What if we want to track and limit the number of credits a student is taking....

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
Objects can have Methods

**Function**: call with object as argument

```python
len(my_list)
```

**Method**: function tied to the object

```python
my_list.count
```

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

Class Folders

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

Complete Class Definition

```python
class <class-name>:
    # <class specification>
    # <assignment statements>
    # <method definitions>
```

Method Definitions

```python
def enroll(self, name, n):
    # <rest of fn definition goes here>
```

More Method Definitions!

```python
for one_course in courses:
    if one_course[0] == course_name:
        self.courses.remove(one_course)
```

## Class Attributes – assign in class definition

```python
class Student:
    def __init__(self, __, netID, courses, major):
        # <specifications go here>
```

### Class Folders

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

### Classes Have Folders Too

- **Class Folders**
  - Data common to all instances

Where does `max_credit` live???

Refer to class attribute using class name

Common to all instances

Recall from class invariant that attribute `courses` is a nested list, so `one_course` here is a list with 2 values: at index 0 is the course name; at index 1 is the number of credits of that course
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)

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")
print(s1.netID)  # finds attribute in object folder
print(s1.max_credit)  # finds attribute in class folder
```

Better to refer to Class Variables using the Class Name

Don’t forget `self`, Part 1

```python
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 (Q)

```python
s1 = Student("xy1234", [], "History")
s2 = Student("ab132", [], "Math")
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`

Better to refer to Class Variables using the Class Name

What gets Printed? (Q)

```python
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)
```