Lecture 18:
More on Classes
(Chapter 17)
CS 1110
Introduction to Computing Using Python

We know how to make:

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

We know how to make:

Review… from last lecture

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
   • in parameter list of method (method header)
   • when defining method (method body)

Don’t forget self, Part 1

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

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

Don’t forget self, Part 2 (Q)

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

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)
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

```
class Student():
    def __init__(self, netID, courses=[], major=None):
        self.netID = netID
        self.courses = courses
        self.major = major

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

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 <,

Optional: for a complete list, see https://docs.python.org/3/reference/datamodel.html#basic-customization

See Fractions example at the end of this lecture

Designing Types

- **Type**: set of values and the operations on them
  - int: (set: integers; ops: +, -, *, /, …)
  - Point2 (set: x,y coordinates; ops: distanceTo, …)
  - Card (set: suit * rank combinations; ops: ==, !=, <, >)
  - Others to think about: Person, Student, 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 instance (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!

Let’s make a word guessing game

- There is a secret word.
- The user has 10 chances to guess letters until the word has been spelled out.
- Would be great to have a class SecretWord that would keep track of both the word we’re guessing and what the user sees / has guessed so far.

Play the game.

How does the game go?

```
word_list = [... candidate words for user to guess ...]
N_GUESSES = 10
Set the secret word

User guesses until no more guesses or secret is solved

Reveal the word
```
What should the SecretWord offer me?

Like a string, but two of them:
1. the secret word
2. what the user sees

I should be able to:
• Set the secret word
• Print out the word as guessed “so far”
• Determine whether the game is over
• Reveal the secret word

Example: SecretWord

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

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?

Planning out Class: the Attributes

class SecretWord:
   """A word to be guessed by a user in a word guessing game."""

   Instance Attributes:
   secret_word: word being guessed [str of lower case letters]
   display_word: word as the user sees it; the letters of secret_word show correctly guessed letters [str of lower case letters and '_']
   secret_word and display_word agree on all letters and have same length

   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

Planning out Class: the Methods

def __init__(self, word):
   """Initializer: creates both secret_word and display_word from word [a str of lower case letters]"

def __str__(self):
   """Returns: both words"""

def __len__(self):
   """Returns: the length of the secret_word"""

def reveal(self):
   """Prints the word being guessed"

def print_word_so_far(self):
   """Prints the display_word"

def apply_guess(self, letter):
   """Updates the display_word to reveal all instances of letter as they appear in the secret_word. ('_' is replaced with letter)
   letter: the user's guess [1-character string in A-Z or a..z]"

def is_solved(self):
   """Returns True if the entire word has been guessed"

   What are the method specifications: states what the method does & what it expects (preconditions)
How is SecretWord to be used?

```python
import random, wordGuess
word_list = [...] candidate words for user to guess ...]

N_GUESSES = 10
Set the secret word

User guesses until no more guesses or secret is solved

Reveal the word
```

Implementing a Class

• All that remains is to fill in the methods. (All?!)  
  • When implementing methods:
    1. Assume preconditions are true (checking is friendly)
    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

Implementing an Initializer (Q)

```python
def __init__(self, word):
    # JOB OF THIS METHOD
    SecretWord.secret_word = word
    SecretWord.display_word = ['_'] * len(word)
A
secret_word = word
display_word = ['_'] * len(word)
B
self.secret_word = word
self.display_word = ['_'] * len(word)
C
```

Instance variables:  
• WHAT HAS BETTER BE TRUE WHEN WE'RE DONE
• secret_word: [str of lower case letters]
• display_word: the letters of secret_word show correctly guessed letters (list of single lower case letters and '_')
• secret_word and display_word agree on all letters and have same length

Implementing apply_guess()

```python
def apply_guess(self, letter):
    # JOB OF METHOD
    letter: the user's guess [1-character string in A..Z or a..z]
    secret_word: [str of lower case letters]
    display_word: the letters of secret_word show correctly guessed letters (list of single lower case letters and '_')
    secret_word and display_word agree on all letters and have same length
```

```python
secret_word: [str of lower case letters]
# WHAT YOU CAN COUNT ON
display_word: the letters of secret_word show correctly guessed letters (list of single lower case letters and '_')
secret_word and display_word agree on all letters and have same length
```
Planning out a Class: Fraction

- What attributes?
- What invariants?
- What methods?
- What initializer and other special methods?

```python
class Fraction:
    """Instance is a fraction n/d"
    Attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
    def __init__(self, n=0, d=1):
        """Init: makes a Fraction"
        assert type(n) == int
        assert type(d) == int and d > 0
        self.numerator = n
        self.denominator = d
```

Problem: Doing Math is Unwieldy

<table>
<thead>
<tr>
<th>What We Want</th>
<th>What We Get</th>
</tr>
</thead>
</table>
| \[
\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \times \frac{5}{4}
\] | \[
\frac{5}{4}
\] |

Why not use the standard Python math operations?

Operator Overloading: Addition

```python
class Fraction:
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
    def __add__(self, q):
        """Returns: Sum of self, q"
        assert type(q) == Fraction
        bot = self.denominator * q.denominator
        top = (self.numerator * q.denominator +
              self.denominator * q.numerator)
        return Fraction(top, bot)
```

Operator Overloading: Multiplication

```python
class Fraction:
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
    def __mul__(self, q):
        """Returns: Product of self, q"
        assert type(q) == Fraction
        top = self.numerator * q.numerator
        bot = self.denominator * q.denominator
        return Fraction(top, bot)
```

Operator Overloading: Equality

```python
class Fraction:
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
    def __eq__(self, q):
        """Returns: True if self, q equal, False if not, or q not a Fraction"
        if type(q) != Fraction:
            return False
        left = self.numerator * q.denominator
        right = self.denominator * q.numerator
        return left == right
```

Optional:
for a complete list, see https://docs.python.org/3/reference/datamodel.html#basic-customization