Lecture 21:  
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

Method Definitions

<table>
<thead>
<tr>
<th>id</th>
<th>Student</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc123</td>
<td>netID</td>
<td>max_credit</td>
</tr>
<tr>
<td>22</td>
<td>courses</td>
<td></td>
</tr>
<tr>
<td></td>
<td>major</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n_credit</td>
<td></td>
</tr>
</tbody>
</table>

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

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

See Fractions example at the end of this presentation

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!
Start next video:
design and implement a
class for a game

---

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?

```python
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

```python
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

```python
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""
```

Are there any special methods that you will need to provide? What are their preconditions? You don’t have to do this. But you should consider it. Careful. Make sure overloading is the right thing to do.

Planning out Class: the Methods

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

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

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
```

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

guess_the_word(secret_word, N_GUESSES)
Reveal the word
```

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

guess_the_word(secret_word, N_GUESSES)
Reveal secret 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

```python
def __init__(self, word):
    '''Initializer: creates both secret_word and display_word from word [a str of lower case letters]'''  # JOB OF THIS METHOD
    Instance variables:  # WHAT 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
    secret_word and display_word agree on all letters and have same length
```

Implementing an Initializer (Q)

```python
def __init__(self, word):
    '''Initializer: creates both secret_word and display_word from word [a str of lower case letters]'''  # JOB OF THIS METHOD
    SecretWord.secret_word = word
    SecretWord.display_word = '_'*len(word)
    secret_word = word
    display_word = '_'*len(word)
    self.secret_word = word
    self.display_word = '_'*len(word)
```

Implementing an Initializer (A)

```python
def __init__(self, word):
    '''Initializer: creates both secret_word and display_word from word [a str of lower case letters]'''  # JOB OF THIS METHOD
    SecretWord.secret_word = word
    SecretWord.display_word = '_'*len(word)
    secret_word = word
    display_word = '_'*len(word)
    self.secret_word = word
    self.display_word = '_'*len(word)
```

Implementing apply_guess()

```python
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) # JOB OF METHOD
    letter: the user's guess [1-character string in A..Z or a..z] # ASSUME TRUE
    lower_letter = letter.lower()
    for i in range(len(self.secret_word)):
        if self.secret_word[i] == lower_letter:
            self.display_word = self.display_word[:i] + lower_letter + self.display_word[i+1:]
```

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
guess_the_word(secret_word, N_GUESSES)
Reveal secret word
def guess_the_word(secret, n_guesses_left):
    if secret is solved:
        print("YOU WIN!!!")
    elif n_more_guesses==0:
        print("Sorry you're out of guesses")
    else:
        print the word-in-progress
        user_guess = input("Guess a letter: ")
        apply guess to the secret word
        guess_the_word(secret, n_guesses_left-1)
```

```python
import random, wordGuess
word_list = [ ... candidate words for user to guess ... ]
N_GUESSES = 10
Set the secret word
guess_the_word(secret_word, N_GUESSES)
Reveal secret word
def guess_the_word(secret, n_guesses_left):
    if secret is solved:
        print("YOU WIN!!!")
    elif n_more_guesses==0:
        print("Sorry you're out of guesses")
    else:
        print the word-in-progress
        user_guess = input("Guess a letter: ")
        apply guess to the secret word
        guess_the_word(secret, n_guesses_left-1)
```
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

What We Want

\[
\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) \times \frac{5}{4}
\]

What We Get

```python
>>> p = Fraction(1,2)
>>> q = Fraction(1,3)
>>> r = Fraction(1,4)
>>> s = Fraction(5,4)
>>> (p.add(q.add(r))).mult(s)
```

Why not use the standard Python math operations?

Pain!

Operator Overloading: Addition

```python
class Fraction:
    """Instance attributes:
        numerator: top [int]
        denominator: bottom [int > 0]
    """

def __add__(self, q):
    """Returns: Sum of self, q
    Makes a new Fraction
    Precondition: q a Fraction"
    assert type(q) == Fraction
    bot = self.denominator * q.denominator
    top = (self.numerator * q.denominator +
           self.denominator * q.numerator)
    return Fraction(top, bot)
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p + q
```

Python converts to

Operator overloading uses method in object on left.

Operator Overloading: Multiplication

```python
class Fraction:
    """Instance attributes:
        numerator: top [int]
        denominator: bottom [int > 0]
    """

def __mul__(self, q):
    """Returns: Product of self, q
    Makes a new Fraction; does not modify contents of self or q
    Precondition: q a Fraction"
    assert type(q) == Fraction
    top = self.numerator * q.numerator
    bot = self.denominator * q.denominator
    return Fraction(top, bot)
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p * q
```

Python converts to

Operator overloading uses method in object on left.

Operator Overloading: Equality

- By default, `==` compares folder IDs, e.g., the following expression evaluates to False:
  ```python
  Fraction(2,5)==Fraction(2,5)
  ```
  - Can implement `__eq__` to check for equivalence of two Fractions instead
  ```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](https://docs.python.org/3/reference/datamodel.html#basic-customization)