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

Lecture 18: Using Classes Effectively

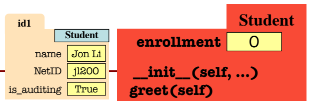
(Chapter 17)

CS 1110

Introduction to Computing Using Python

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

Method Definitions



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

Example: p1.greet()

- Go to class folder for p1 (i.e., Student) that's where greet is defined
- Now greet is called with p1 as its first argument
- This way, greet knows which instance of Student it is working with

```
class Student():
    def __init__(self, name, NetID, is_auditing):
        self.name  = name
        self.NetID = NetID
        self.is_auditing = is_auditing
        Student.enrollment = Student.enrollment + 1

    def greet(self):
        """Prints information about the
        Student to the screen"""
        print("Hi! My name is "+ self.name)
        print("My NetID is "+ self.NetID)
        if self.is_auditing:
            print("I'm auditing the class")
```

Special Methods in Python

- 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()

__repr__ for repr()

__eq__ for ==, __lt__ for <, ...

- For a complete list, see

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

```
class Point3():
    """Instances are points in 3D space"""
    ...

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes new Point3"""

        ...

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

    def __repr__(self):
        """Returns: unambiguous string"""
         return str(self.__class__) + str(self)
```

See Fractions example at the end of this presentation

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**
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!

Planning out Class: the Attributes

```
class SecretWord(object):
  """A word to be guessed by a user in a game of hangman.

  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"""
```

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

```
def word_so_far(self):
    """Prints the word being guessed"""
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 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 this going to be used?

```
import random, hangman
word_list = [ ... words we want user to guess .. ]
N_GUESSES = 10
secret = hangman.SecretWord(random.choice(word_list))

for n in list(range(N_GUESSES)):
   secret.word_so_far()
   user_guess = input("Guess a letter: ")
   secret.apply_guess(user_guess):
   if secret.is_solved():
         print("YOU WIN!!!")
         break  #jumps out of the for-loop, not allowed for A3!
secret.reveal()
```

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)

```
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
A   SecretWord.display_word = len(word)*'_'

    secret_word = word
B   display_word = len(word)*'_'

    self.secret_word = word
C   self.display_word = len(word)*'_'
```

```
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
                [str of lower case letters and '_']
  secret_word and display_word agree on all letters and have same length
```

Implementing guess()

```
secret_word: [str of lower case letters]      # WHAT YOU CAN COUNT ON
display_word: 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
```

```
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 A..Z]"""        # ASSUME TRUE
```

```
secret_word: [str of lower case letters]      # WHAT STILL BETTER BE TRUE
display_word: 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
```