PRACTICE CLASS IMPLEMENTATION: BANK ACCOUNT

Review

- What is our process for creating a class?

Review: Process for Creating a Class

- Determine data, functionality
- Create class
  - Create __init__, __str__ methods
- Test those methods
- Repeat: create a method, test

Bank Account

- Data
  - Account name/id
  - Account balance
- Functionality
  - Constructor
  - String
  - Getters for the account info (e.g., id and balance)
  - Deposit money
  - Withdraw money

Consider what these methods should take as parameters

PRACTICE PROBLEM:
COURSE MANAGEMENT SOFTWARE

Our Goal: Replacement for Workday!

- We want to keep track of courses and enrollment in those courses
  - Later: keep track of the enrolled students

What is our process for developing a class?

Class: Course

Part 1

- Data
  - Name
  - Max course capacity
  - Number of students enrolled in the course
- Functionality
  - Constructor: Course(name, max_course_capacity)
    - Why does it not take the number of students enrolled in the course?
    - How will you handle that in the constructor?
  - Str format: `<coursename>: <enrolled> out of <capacity>`
  - Register and drop students from the course (1 at a time), e.g.,
    - addStudent(), dropStudent()
  - Check if course is full
  - Find out how many seats left

Creating a "basic" class should be getting easier with more practice.
Continue to the next part OR create another basic class, like, Bank Account for more practice

Class: University

Part 2

- Keeps track of the Courses in the university
  - How? How do we model this information?
- Functionality
  - Displays the courses in a nice way
    - 1st draft: for each course, show: id name enrolled seatsleft
    - 2nd draft: Put into a nicely formatted table
  - Create/adds courses, removes courses
  - Add/remove students from a course

Interface: Workday2.0

Part 3

- Create an interface for the University
- Allows users to
  - View a course
  - Add/drop students from courses
  - …

Emphasis on *using* the University and Course classes/objects/API

Iterate!

Part 4

- You can probably envision a lot of additional functionality for each part
  - Add it and test, one part at a time!

Course Management Software – the Next Generation

- Create a Student class
  - Has an id, name, list of course ids
  - (can add more state/functionality for more practice)
  - Functionality: constructor, string, __eq__ method so that adding/removing students is easier
- Modify the Course class to add and remove students by taking a Student object
  - How will this affect how you determine the number of enrolled students?

PRACTICE PROBLEM:
ANIMAL SHELTER SOFTWARE

Animal Shelter Software

- We want to keep track of animals at an animal shelter

What is our process for developing a class?

Class: Pet

- Data:
  - Species of animal (dog, cat, chinchilla)
  - Name
    - Defaults to “”
  - Status (in holding, in adoption room, adopted)
    - Defaults to “in holding”
- Functionality
  - Constructor: `Pet(species)`
  - String format: `"species: name, status"`
  - Setters for name
  - Set animal as adopted or in adoption room
  - Getters for this information

Where do we go from here?

- Hopefully, you're understanding how we develop software. Now that you have the Pet class, what can you do next?

MORE PRACTICE PROBLEMS (NOT ONLY CREATING CLASSES)

Counter Class Specification

- Implement, Test
- Example use: Caesar cipher

- A class that represents a counter that wraps around from a high value back to its low value
- Data:
  - Low, high, and current values (all integers)
- Functionality:
  - Constructor – takes as parameters the low value and the high value
    - counter starts at low value
  - A string representation of the Counter
    - Format: "low: <low> high: <high> current: <current>"
  - Getters: low, high, current value
  - Increment the counter by a given amount (a positive amount), wrapping around to low again, if necessary. Returns number of times had to wrap around.
    - Example: if counter's low is 0 and the high is 9 and its current value is 9:
      - `test.testEqual(counter.increment(1), 1); test.testEqual(counter.getCurrent(), 0)`
  - Decrement the counter by a given amount (a positive number), wrapping around to high again, if necessary. Returns number of times had to wrap around.
  - Set the counter's value, only if low <= value <= high. Otherwise, displays an error message.

Palindrome

- Write a program that determines if a string (input by a user) is a palindrome. A *palindrome* is a word that is the same forwards and backwards. Some example palindromes: "kayak", "A man A plan A canal Panama".
- http://www.fun-with-words.com/palin_example.html
- Break the problem into at least two functions:
  - main
  - isPalindrome, which returns True iff the parameter string passed into the function is a palindrome.
- Depending on how you think about the problem, you may want to break the solution into more functions, e.g., a reverseString function

Generate a Random Password

- Function: given number of characters
- Returns a random password
  - Includes upper, lowercase letters; numbers; punctuation

Function: createDict

- Write a function that, given two lists of equal length
  - The first list is the keys
  - The second list is the values
- Returns the dictionary that maps the keys from the first list to the values in the second list, respectively/in order
- Examples:
  - test.testEqual(createDict([1, 2], [“one”, “two”]), {1:”one”, 2:”two”})
  - test.testEqual(createDict([1, 2], [“two”, “one”]), {1:”two”, 2:”one”})

Fibonacci

- Solve the Fibonacci sequence *recursively*
- Note: this is less efficient than the iterative solutions you wrote during lab