Midterm Review

Winter 2016

Recap of big ideas and Python topics

Exam Format

The exam will be closed book / closed notes

- a set of reference pages with names of Python functions, GUI elements, etc will be provided

50 minutes

100 points

20% of course grade

Encapsulation

One of the “big ideas” in CIS 211

benefits:

- reuse
- modularity
- getting organized (team projects as well as personal to-do list)

information hiding

- specific form of encapsulation
- implementation details are “hidden” inside module/class
- users interact with code through API (application program interface)

Encapsulation Example: Room

API:

```
>>> from Room import *
>>> x = Room('kitchen', 15, 20)
>>> x.width()
15
>>> x.area()
300
```

From the outside we just know rooms have names and dimensions and that we can get these attributes by calling methods.

Possible implementations:

`def __init__(self,n,w,d):`
`  self._name = n`
`  self._width = w`
`  self._detph = d`
	`def __init__(self,n,w,d):`
	`  self._name = n`
	`  self._dim = (w,d)`

`def area(self):`
`  return self._width *`
`  self._depth`
	`def area(self):`
	`  return self._dim[0] *`
	`  self._dim[1]`

The person who implements the class decides what instance variables are required to implement the specified behavior

Encapsulation Example: Venue

Suppose we are working on a project for a ticket broker

The programs might use a class named Venue to represent a location where events are held (stadium, club, arena, …)

The API might include:

```
>>> x.name()
'Autzen Stadium'
>>> x.capacity()
53800
>>> x.reserve(2)
[(11, 20, 7), (11, 20, 8)]
```

The implementation might include:

- instance variables for the name, other simple attributes
- a list or dictionary of sections, where each section is a list of rows and each row is a list of seats
- other data structures (e.g. a list of premium sections)

Important point: how the seats are organized is a decision made by the implementer.

Code that uses the object calls methods to find available seats, etc

Python: Namespaces

Python implements OOP concepts (encapsulation and information hiding) through the use of ***namespaces***

- top level program (or interactive session)
- a new temporary namespace is created when a function is called (initialized with arguments, local vars are added to this namespace) [aside: recursion]
- modules (static collection of names of functions, vars, …)
- classes (collection of names of methods and class variables)
- object instances (each instance is a namespace linked to a class)

How we look at namespaces: `type(x)` and `dir(x)`

How namespaces are modified:

- assignment (introduce new var)
- `def` (introduce new function)
- `class` (introduce new class)

Note: Python does not have true information hiding

- in some languages the compiler prevents code outside a class from accessing methods or variables defined inside a class
- programming conventions and naming standards help programmers write OOP
- if names are intended to be “private”:
    - instance vars: names that start with a single underscore
    - class vars: names begin with (but don’t end with) double underscore

Python: Classes

A `class` statement introduces

- a new type
- a function that can be called to create objects of that type

Functions defined inside a class become ***instance methods***

Python transforms a statement like

```
x.area()
```

into

```
Room.area(x)
```

When we write the method definition we include an additional parameter name

- by convention the parameter is named `self`
- when the method is called `self` is a reference to the object

Class Variables and Class Methods

Variables defined inside a class become ***class variables***

There is one copy of the variable, shared by all instances

Refer to the variable using the qualified name syntax, e.g.

```
GiftCard.csym
```

We can also define ***class methods***

- put the “decorator” `@staticmethod` before the `def` statement
- tells Python not to do the transformation shown above, i.e. the method is not expected to work on single instances
- typically used for functions defined for the whole class

```
@staticmethod
def set_currency(sym):
    GiftCard.csym = sym
```

Special Functions

Some functions (all with names that begin and end with two underscores) are called automatically in special circumstances

If `x` is an object of a class, these methods are called

`__init__`	when an object is created
`__lt__`	when `x` is compared to another object
`__repr__`	when Python needs a string to represent `x`
`__getitem__`	when Python evaluates `x[i]`
`__setitem__`	when `x[i]` is on the left side of an assignment
`__len__`	when Python evaluates `len(x)`
`__add__`	when Python evaluates `x + y`
`__sub__`	when Python evaluates `x - y`

Note: the exam will include a reference sheet, you don't need to memorize these names (but know what the functions do, when they are called)

Inheritance

A special form of encapsulation, emphasizes reuse

- "an X is a Y"

Python syntax: name of parent class (aka "super class" or "base class") enclosed in parens after new class name, e.g.

```
class PhoneCard(GiftCard):
```

If `x` is an instance of class C and we call `x.foo()` Python

- checks to see if `foo` is defined in C
- if not, look for `foo` in C's parent class
- continue working up the hierarchy (all the way to `object`, if necessary)

Code in the derived class can define functions with the same name as those in the base class

- the new definitions override the parent class methods
- the name lookup mechanism makes sure derived class methods are found before the base class methods

OOP Examples

Geometric shapes (Shape / Polygon / Square, ...)

GiftCard / PhoneCard

Room / Atrium

list / Queue (new class derived from a Python built-in class)

Graphical User Interfaces (GUI)

A GUI consists of a top level window and an associated set of widgets

We used a standard Python library called `tkinter`

A `tkinter` app has

- a top level window
- widget objects instantiated from classes named Button, Entry, Frame, etc

Widgets are organized in a hierarchy

When a widget is created the constructor is passed a reference to the enclosing class

```
hello = Label(app, text = 'Hello, world')
```

To make a widget visible on the screen we need to call a ***layout manager*** (we used `pack` and `grid`)

Labels

Simple widgets, used to display text or an image

Usually static, but they can be changed, e.g.

```
hello['text'] = "Bonjour tout le monde"
```

Buttons

Used to carry out some operation when the user clicks on the button, e.g.

```
fr = Button(root, text = 'Français', command = fr_cb)
```

Here `fr_cb` is the name of a callback function — a function with no arguments that will be called when the button is clicked

Entry

A simple widget for getting information from a user

Users enter text, methods access the contents as a Python string

Editing operations can modify the text (erase chars, insert new chars, …)

Miscellaneous Python Topics

There may be questions on

- lists and dictionaries
- list comprehension
- passing functions as arguments to other functions
- other Python language constructs covered in class

Style Points

If you are asked to fill in the body of a function use good style

- good choice of names
- standard names (e.g. `self`)
- short, concise statements

If there are any programming questions they will be very short (the amount of space you are given is a strong hint)

What to Expect (OOP)

There will be questions on Python classes

- what is the value of call to `Foo.x()`?
- explain what happens when ....
- add a method to the class that ... (maybe give header, you fill in the body)
- give the outline of a class that ... (show the __init__ function, names of methods, ...)

Some questions will be based on a new kind of object (similar in complexity to GiftCard or Room)

What to Expect (GUI)

The exam will include a working GUI, similar in complexity to the temperature converter or “Hello, World” with frames

Questions will ask “what happens when...” or “what is the result of calling...”

You may be asked to add a widget or write a callback function

Reference Sheet

There will be a reference sheet — don’t worry about memorizing syntax, names of special methods (`__lt__`, …), names of GUI elements, …

Show Your Work!

Don’t just write a number or “yes” or “no” — explain why you think your answer is correct

Sneak Preview

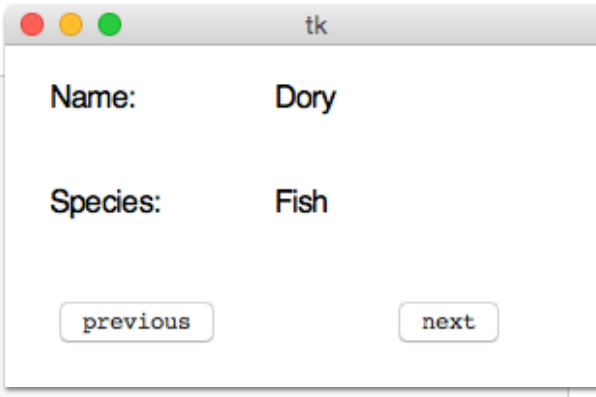
The class hierarchy used for exam questions is based on kinds of animals.

- attributes include name, skin color, etc

A GUI based on this class is shown below. Widgets are:

- labels that show name and species
- buttons that show attributes

The program has a list of animal characters from a movie. Clicking the 'previous' and 'next' buttons updates the display to show the previous character or the next character




```
class CharacterFrame(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.pack()

        Label(self, text='Name:').grid(row = 0, column = 0)

        self._name = Label(self, text = '')
        self._name.grid(row = 0, column = 1)

        # ... other labels and buttons defined here ...

        self._next = Button(self, text='next', command=self.next_character)
        self._next.grid(row=2, column=1, pady=20)

        self.set_character(0)

    def set_character(self, n):
        self._character_num = n
        self._name.config(text = characters[n].name().capitalize())
        self._species.config(text = characters[n].species())

    def next_character(self):
        if self._character_num < len(characters) - 1:
            self.set_character(self._character_num + 1)
```