CIS 211 Final Review

Winter 2016

Recap of big ideas and a guide for studying for the final exam

Last page of this document: general hints and strategies
(e.g. don't memorize, there will be a reference sheet)

Course Summary

Two goals for CIS 211:

- more experience with programming, especially with Python
- introduce fundamental concepts in CS

A "spiral curriculum" has a broad introduction, then a set of more advanced courses that build on the introduction

Why Programming? Why Do CS Majors Write Programs?

A CS degree from a college or research university is not the same as taking classes at "trade tech"

But it does require students to learn to program, and students write *lots* of programs

- hands-on experience with algorithms and problem solving techniques
- "learning by doing"
- the same reason math students solve equations, English students write essays, chemistry students do lab projects: not just "absorbing" material, but making decisions, carrying out projects

"Education is what's left after you have forgotten everything you have learned."

Why Python?

Simple language, easy to get started

Support for object-oriented programming

⋆ Interactive

⋆ IPython / Jupyter Notebooks

Again, goal is not (just) to become proficient Python programmers, but to use Python to experiment with CS topics

Abstraction

A unifying theme throughout the term, can be seen in all the topics we covered

A requirement for dealing with complicated systems

Ways abstraction helps deal with complexity:

encapsulation — if I look in the garage I might see 4 tires, a couple of seats, a steering wheel, and a bunch of blue metal, but it's useful to think of the combination as a single item: a car

- related to "chunking" in psychology ("magic number 7")

layering — to use the car I insert a key, push a button, push another button to lower the top, and operate pedals and a steering wheel, aka the "user interface"

- I don't need to worry about how these things are implemented (the electric circuit closed when I push a button, the amount of fuel injected into a cylinder when I press the gas pedal, …)

Levels of Abstraction

What do we mean by "high level", as in "high level language"?

Answer: "level" is related to distance from machine (hardware)

Low level languages (machine language, assembly language) require programmers to know details about the CPU (names of registers, types of instructions that execute on a single fetch/decode/execute cycle)

Some languages described as "HLL" are really intermediate level: C

aside: `register` is a keyword in C...

Higher (but still have some notion of the underlying machine): Java, C++
(no "garbage collection", programmer manages storage)

Even higher: Python, Ruby, other machine-independent OOP

Highest: FP, SQL, others ("non von Neumann") that aren't based on control flow; ***declarative*** instead of ***imperative***

Abstraction in Computing I: Libraries

Introduced in CIS 210, heavily used in CIS 211

Modules are collections of related data and functions

`math`: defines `pi` and other values, as well as `sqrt` and other functions

(and note the abstraction at work: we don't need to know the algorithms used to compute the square root of a number, we just use the "package")

`tkinter`: functions for creating, using "widgets"

In Python:

- variable names are organized as ***namespaces***
- functions need to be ***imported*** from a library before we can use them
- names in Python are simply ***references*** to objects, which can be data (numbers, strings, instances), functions, classes, or modules
- a ***qualified name*** (`x.y`) includes of one or more module/class names (*e.g.* `os.path.join`)

Examples:

```
>>> import re
>>> import tkinter as tk
>>> from math import cos
>>> from math import pi as π
```

Abstraction in Computing II: OOP

A ***class*** is a collection that contains

- variables that define the state of an object (fuel level, top up or down, …)
- functions (aka ***methods***) that do computations based on the state and/or update the state

Individual objects are ***instances*** of the class

An important type of abstraction in OOP: ***inheritance***

We can define a new class by building on definitions from existing classes

> "an X is a type of Y except for …."
>
> "a convertible is a type of car that …"
>
> "a PhoneCard is a type of Card that can be 'topped up' "
>
> "a Queue is a list that …."

In Python:

- a `class` statement defines a new type of object
- `class X(Y)` says the new type X is derived from an existing type Y
- the name of a class becomes the name of a function we can call to create objects of the type (a ***constructor***)

Expect questions related to defining and using classes in Python

- know the basic syntax
- know how to define methods inside a class, and that the first argument passed to a method is a reference to an object (`self`)
- know how `__init__` and other special functions are used

The class hierarchy for the exam (DNA) is in an IPython Notebook called DNA Demo

Abstraction in Computing III: GUI

Graphical User Interfaces (GUIs) are a perfect example of how abstraction helps organize programs

Apps with GUIs are ***very*** complicated

Even “hello world” has a surprising amount of complexity:

- is the display updated when the button is pushed?   or when it is released?
- can the user press the button, then change their mind by moving the mouse before releasing it?

All of these details are encapsulated inside objects defined in the ***widget*** classes

Modern applications (web browsers, spreadsheets, games, …) are only possible through the use of well-organized layers of abstraction

In Python:

- basic widgets are defined by classes in the `tkinter` library (Button, Label, …)
- geometry managers (`pack`, `grid`) place widgets inside an enclosing (parent) widget
- ***callback functions*** are attached to widgets, carry out computations in response to user actions

Expect questions related to `tkinter`

Given an example program, be able to explain

- what it will display
- what will happen if a button is clicked
- how to modify the program to change/add behaviors

The GUI for the exam is also in the DNA Demo notebook

Abstraction in Computing IV: SQL

A relational database is another good example of abstraction at work

We can think of a RDB as a separate module

- a “class” that implements a type of data called a ***table***
- a set of functions that create, access, and update tables

An RDB is a collection of tables with named ***columns***.

Each ***row*** in a column represents a separate piece of data (aka a ***record***).

In practice the RDB is mainly used to store ***persistent*** data (data that lives on disk, outside the program, so it survives after the program exits)

SQLite is a RDB that keeps data in a file in the user’s directory

- MySQL, others use a client-server model; a client application connects to a server, which manages the data at an external location

SQL is a High Level Language

Another aspect of abstraction: SQL (the standard language for working with RDBs) is a very high level language

- verbose, arcane, but still “high level”
- specify ***what*** you want from the DB, not ***how*** to accomplish the task
- the system will compile the statement into a sequence of actions that carry out the request

Databases and Python:

- use functions from a module named `sqlite3` to access data in a SQLite database
- pass a SQL query to `execute`, get back an iterable object that will produce rows from the query result

Expect SQL and sqlite3 questions

- be able to explain what a query will return, or how to write a query based on a simple description (“all actors who starred in X”)
- understand how a ***join*** operation is required to get information from more than one table
- questions may be related to `sakila211.db` or another simple DB

The SQLite database for the exam (Foodila) is also demonstrated in DNA Demo

Abstraction in Computing V: Functional Programming

From a Python programmer's perspective, "functional programming" means "***programming with functions***"

A set of builtin functions take a reference to another function as an argument and ***apply*** the function to each item in a collection

This is sometimes known as "procedural abstraction"

In Python:

`map(f,x)` returns an iterable containing $f(x_i)$ for each item in `x`

`filter(f,x)` returns an iterable that contains all x_i such that $f(x_i)$ is true

`reduce(f,x)` returns the result of applying `f` to pairs of items from `x`

Examples:

```
>>> a = ['alpha', 'beta', 'pi', 'upsilon']

>>> m = map(len, a)

>>> list(map(len,a))
[5, 4, 2, 7]

>>> list(filter(lambda s: s.find('p') != -1 , a))
['alpha', 'pi', 'upsilon']

>>> from functools import reduce
>>> from operator import add
>>> add(1,1)
2
>>> reduce(add, map(len,a))
18
```

Expect questions on functional programming

- describe the result from evaluating an expression containing a call to `map`, `filter`, or `reduce`
- write an expression that uses `map`, `filter`, or `reduce`

Abstraction VI: Simulation

Two of our projects this term were computer simulations

- solar system (simple example of an N-body system)
- bears and fish (“Game of Life” style cellular automaton)

Simulation is yet another example of abstraction

- what are the important attributes of planets?
  - yes: mass, location, velocity
  - no: color, atmosphere, date discovered, inhabitants
- what are the important attributes of bears? fish?
  - yes: location, mobility, feeding, breeding
  - no: color, size, male/female, habitat, ...

If we were working on a more realistic model we would probably revisit some of these decisions

- a model needs to be as complex as necessary, but not more
- if the model does not make accurate predictions go back and reconsider attributes and behaviors

Expect questions on vectors, equations of motion, cellular automata

Designing With Abstractions

When faced with a complicated project, look for patterns, and look for ways to "compartmentalize"

- find parts that can be separated and implemented as "subroutines" (functions)
- can similar pieces be collected into a module, or be used to define a new type of object?

This term: introduction to OOP concepts, using OOP constructs when writing programs

In the future try to "be lazy"

- in general, avoid control structures (`if`, `for`, ...) if there is a data structure that will simplify the program
- instead of writing a loop to do an operation is there a way to use `map` or some other FP construct?
- is there a library module that implements the operation?
- don't forget that SQLite tables are an option for organizing tables "in memory"

General Advice

Be Familiar with Python Objects and Their Methods

```
s = 'hello'
s.capitalize()
s[0]
s[1:4]

a = ['h', 'e', 'l', 'l', 'o']
a[1:4]
a[1:4] = 'u'
a[-1] = 'm'
''.join(a)
```

Don't Memorize

There will be a reference sheet

- commonly used methods (`str.upper`, `str.strip`, …)
- names of important methods to overload when defining classes (`__init__`, `__repr__`, …)
- modules we used and important functions in those modules (e.g. `execute` in `sqlite3`, `reduce` in `functools`, …)
- GUI classes (Button, Entry, …)

Show Your Work

Explain how you got an answer