CIS 211 Introduction to Computer Science* II

• this is the middle course in our 3-quarter first year sequence
• we will have a bit of time to reflect on the “S” word in “Computer Science”
• but first, let’s get down to what this course is about
  – homework: (programming) 30%
  – 2 midterm exams: 15% + 15% (2/11 and 3/11)
    (don't miss them; no early exams, no makeups!)
  – 2 quizzes: 5% + 5% (unannounced, no makeups)
  – final exam: 20% (Monday 3/18/2013 at 1515)
    (don't miss it; no early exams, no makeup!)
  – commentaries: (blog entries) 10%
• GTFs
  – Azad Abbasi
  – Daniel Ellsworth
• the required texts
  – Introduction to Computing Using Python (same as in CIS 210)
  – Head First Design Patterns (purchase or available online via library)

science? science? we don’t neeeeed no ...
CIS 210 — 211 — 212

• CIS 210 introduced Python, a modern and viable programming language
  – you learned elements of syntax and underlying concepts such as bindings
  – you learned algorithmic problem solving and procedural decomposition
• CIS 211 is a continuation of the topics initiated in CIS 210
  – deepening your understanding and skills in programming in Python
  – progressing further into Introduction to Computing using Python
  – and working through much of Head First Design Patterns (HFDP)
    HFDP uses Java to illustrate the patterns; we will explore how to adapt these patterns to Python, and in the process, learn about Java and how it contrasts with Python
    HFDP available online (go to UO library’s proxy server)
• CIS 210 and 211 together provide a broad introduction to programming
  – useful to non CS majors
  – foundational for CS majors
• for those of you continuing on to CIS 212 Introduction to Computer Science III
  – CIS 212 introduces core concepts of data structures and algorithms
  – the programming language will shift to Java (also used in CIS 313)
creating software from scratch using design patterns

• you will be developing a significant programming project this term
  – a game, or simulation
  – written in Python 2.7 (not 3.x)

• PyCharm:
  – a real, industrial-strength, IDE for Python
  – you will have an educational license (for non-commercial use)

• ‘staged delivery’ term project
  – conceptualizing your own problem domain
    what active agents, inactive objects, actions, locales, events?
    classes and methods that capture essence of that domain
  – using Unified Modeling Language (UML) to describe the entities
    comprising your domain, and their relationships

• progressive refinement, elaboration of your OOD as you develop an
  implementation using OOP techniques
  – basic programming practices
  – successively incorporating design patterns
the textbooks: necessary but not sufficient

- **The Python Tutorial:** [http://docs.python.org/2/tutorial/](http://docs.python.org/2/tutorial/)
- **PyCharm:** [http://www.jetbrains.com/pycharm/quickstart/](http://www.jetbrains.com/pycharm/quickstart/)
- **Patterns in Python:** [https://github.com/faif/python-patterns](https://github.com/faif/python-patterns)
- **First of all... you can search the documentation:**
  - [http://docs.python.org/2/search.html](http://docs.python.org/2/search.html)
- **Intermediate and Advanced Software Carpentry in Python**
  - [http://ivory.idyll.org/articles/advanced-swc/](http://ivory.idyll.org/articles/advanced-swc/)
- **Code like a Pythonista: Idiomatic Python**
- **Regarding modules:**
  - [http://effbot.org/zone/import-confusion.htm](http://effbot.org/zone/import-confusion.htm)
- **Thinking in Python** (which downloads to a useful local site and example code!):
  - [http://www.mindview.net/Books/TIPython](http://www.mindview.net/Books/TIPython)
- **How to Think Like a Computer Scientist:**
  - [http://openbookproject.net/thinkcs/python/english3e/](http://openbookproject.net/thinkcs/python/english3e/)
Hierarchies

Making some hierarchies first:

**Animate** objects
entities that are decidedly active, not passive, can move between **Locations**, engage in actions with **Inanimate** objects and each other.

**Inanimate** objects
things that are acted upon, have **Location**, are passive (or reactive) more than active.

**Location** objects
where things **Actions** happen that involve **Animates** and **Inanimates** (or other **Animates**).

**Action** objects
something that happens involving **Animates** and **Inanimates** (or other **Animates**) at some **Location**.
Creating hierarchies is good, but runs into limitations:

- Good for creating a simple collection of subclasses, each with different behaviors and attributes, with much shared behavior in the superclasses,
- But if the subclasses have divergent behaviors associated with similar named methods, ...

The text *Head First Design Patterns* will show how to create far more elegant and extensible code than might come to mind just using standard OOP practices:

- The Java code reflects the static-typed nature of that language
- The implementation decisions in the book are often forced by the language
- In Python, functions are objects, which influences our design decisions
to OO, and beyond!

• creating fresh new programs from scratch, using the very finest of modern Object Oriented ingredients:
  – the basic classes subclasses, encapsulation, shadowing, exceptions, etc.
  – and want subclasses to differ in their “behavior” (Strategy Pattern)
  – and to have objects “aware” of one another (Observer Pattern)
  – add functionality across a hierarchy without disturbing it (Visitor Pattern)
  – give objects a sense of state-dependent behavior (State Pattern)
  – dynamically change their attributes (Decorator Pattern)
  – ... and so forth

• plus careful reflection on the design as it evolves
  – rapid prototyping and
  – refactoring (not necessarily a bad thing)
what is the nature of a scientific result in computer science?
what is the vocabulary?
what sorts of concepts are expressed?
some 200-level computer science concepts

- computation (at least, to computer scientists)
  - process versus processor
    - von Neumann architecture and other models
    - a language and its interpreter
    - interpretation as process
  - representations
    - numbers, objects, and relationships
    - algebraic and logical expressions
    - algorithm and flow of control
    - sequentiality and parallelism
200-level concepts, con’t.

• algorithm *complexity* and the order of growth
  – big "O" notation, e.g., $O(n)$, $O(n^2)$
    recursion (linear, tree, ...)
  sequentia**lity, parallelism, search, heuristics, hard problems

• *simulation*, *models*, and *abstractions*
  – abstracting processes, interactions between objects
  – modeling natural systems
200-level concepts, con’t.

• **encapsulation** (of data, methods)
  – what do we mean by “data” versus "methods"?
  – is that a hard and fast dichotomy?
  – why is encapsulation useful?

• **instance / object / class**
  – what sorts of concepts are these?
  – how do they relate to their real-world counterparts?
  – *computer science makes abstraction concrete*

• **representations**
  – of abstract data structures (ADTs)
    (e.g., arrays, stacks and queues, trees, ...)
    what sorts of things are they?
  – of domain-specific things, relationships
211 concepts

- *Object Oriented Design*
  - objects and encapsulation
    - division of labor (responsibilities of objects)
    - abstraction barriers
  - abstract engines
    - state machines (finite state automata)
    - interpreters
  - designing for extensibility, maintainability (and other *ity’s*)
    - capitalizing on similarities
      - solve once and reuse, ...
  - design patterns
    - state, visitor, observer, decorator, ...
  - containers and enumeration
    - encapsulating iteration
211 concepts, con’t.

- **Object Oriented Programming**
  - classes, methods and instance variables
  - class hierarchies and simple inheritance
  - controlling access through visibility modifiers
  - shadowing of variables, methods
  - abstract classes and interfaces
  - exceptions
  - graphics and event listeners
  - ‘best practices’ in software development (testing, delivery, ...)


relationships between classes

• a basic relationship: **membership** (as defined by shared properties)
  – (informal usage in English phrases “*x is a kind of …*”, or simply “*x is a …*”)
    a Human is a **MetabolizingThing**
    a Human is a **AnimateThing**
    a Human is a **ThingWithMass**
    a Human is a **BiodegradableThing**
    a Human is a **LargerThanBreadboxThing**

• some such set membership relationships naturally nest:
  – an A is a B (which is a C (which is a D …)
    if x is an instance of A, then x also has the properties of being B, …
  – i.e.,
    a Human is a
    Primate is a
    Mammal is a
    Vertebrate is a
    Animal ...

• “has the properties of” = **inheritance**
inheritance of properties

- Human inherits
  - its grasping hands from Primate
  - its fur from Mammal
  - its backbone from Vertebrate
  - and its metabolism from Animal.
- define the base class (in this case Animal), then extend with subclasses
- the child class inherits most (but not all) the properties of the parent class
- simple versus multiple inheritance
  - a strict tree structure derives from having a single parent
  - the single parent usually has multiple children
  - while biologically, each child usually inherits from two parents
    “ah, he has his father’s eyes and his mother’s nose… isn’t he just adorable?”
  - having two parents is termed ‘multiple inheritance’ in programming
- but in everyday life object derive their properties from many more sources:
  - objects are members of multiple, orthogonal class hierarchies
    e.g., Humans and Refrigerators (what’s in common?)
- [we'll get to that complexity later; we'll stay with simple hierarchies for now]
Python and Java supports hierarchies of classes

- two benefits to creating a class hierarchy
  - increasing functionality by specialization through subclasses
    - e.g., evolving a Swiss Army Knife from a simple pocket knife
      significant design commonality (attributes, properties)
      multiple shared functions (open, close, attach to belt, …)
    - elaborations not necessarily shared
  - modifying attributes to create a suite of variations through subclasses
    - e.g., denominations of currency, font styles, …

- a simple hierarchy is effective:
  - when sibling subclasses exhibit regularity
  - when features or capabilities increase monotonically with depth
    (it is not so easy to subtract capabilities/attributes in subclasses)

- most real hierarchies have exceptions
  - attributes/properties do not always apply across subclasses
  - some features are lost in derived subclasses (think penguins)

- Python provides for multiple inheritance directly
- Java attempts to provide for multiple inheritance through interfaces
Python and Java supports hierarchies of classes

- in Python
  class Animal(object):
      … place the basic functions and attributes of an animal here

  class Vertebrate(Animal):
      … place the added functionality specific to vertebrates here

- in Java
  class Animal extends Object {
      … place the basic functions and attributes of an animal here
  }

  class Vertebrate extends Animal {
      … plus the added functionality specific to vertebrates here
  }
Java only supports simple class hierarchies

- each class in a hierarchy may extend only a **single** parent class
  ```java
class Cat extends Mammal { <additions> }
class Dog extends Mammal { <additions> }
```
- variables in Java are **Statically typed** (explicitly declared to be of a given type)
  ```java
  Mammal m;
  Cat c = new Cat();
  Dog d = new Dog();
  
  // in Java, one may assign an instance of a Cat to a variable of type Mammal
  m = c;
  
  // then both of these expressions are true in Java
  m instanceof Cat
  m instanceof Mammal
  ```
- variables in Python are **not** typed, but **objects** do have type.
  ```python
  class Cat(Mammal): ...
  m = Cat()
  
  // then both of these boolean expressions are true in Python
  isinstance(m, Cat)
  isinstance(m, Mammal)
  ```
variables, types, and pronouns

- in Java, a variable has an explicitly-declared **type**
  - and can only be assigned *(refer to)* instances of objects of that type
    
    ```java
    Apple a = new Orange();
    ```

- similarly, in English, the referent of a noun or pronoun must ‘make sense’
  - “that boat” must refer to some kind of boat
  - “I’m a whale” [H. Simpson] is only metaphorical

- in Python, variables are only **names**, and can refer to anything
  
  ```python
  apple = Orange()
  currentTime = Hippopotamus()
  ```

- in Java, a variable need not refer to anything (its value defaults to **null**)
  
  ```java
  Apple apple;
  Apple apple = null;  // same as previous line
  ```

- in Python, a variable must refer to something (or be explicitly assigned **None**)
  
  ```python
  apple = None
  ```

- Python is **dynamically typed**, while Java is **statically typed**

- [Python and Java are both **strongly typed** (an issue regarding expressions)]

- so consider variables, pointers, pronouns, and physically pointing at an object
  - so CS explicitly deals with ideas of naming, reference, and type
**DYNAMIC TYPING IS GOOD**

In a dynamic typed language, you don’t have to initialize variables, which is a big bonus for many developers. Programmers like the fact that you can use a variable at will when required (without having to initialize it). Dynamic typing is characteristic of many of the scripting languages: Perl, PHP, Python, etc. Dynamic typing, in fact, does save you from writing a few "extra" lines of code, which, in turn, means less time spent writing code.

**OR IS IT?**

The very characteristic of dynamic typed languages that appeals to many developers is also a pitfall, and a major one at that. Consider the following simple example:

```python
/* Python code */
my_variable = 10
while my_variable > 0:
    i = foo(my_variable)
    if i < 100:
        my_variable++
    else
        my_varaible = (my_variable + i)/10 // spelling error intentional
```

As you can see in the above code, `my_varaible` is a spelling mistake that the programmer could have very well made. The problem here is that, since Python is dynamically typed, it will not return an error, but instead will create a new variable called `my_varaible`. So, now we have two variables: `my_variable` and `my_varaible`. This obviously is a serious problem; some would suggest that forced variable declaration is an important requirement in any programming language.
inheritance and class hierarchies

• Python doesn’t make a big deal about types (until it is too late)
  – variables are just labels associated with objects
    common metaphor: a name tag attached to an object
    the name tag has no type, but the object does
  – you might want to know what type of object is referred to by a variable:
    ```python
    if type(v) is Dog
    ```
  – or alternatively, given a class hierarchy
    ```python
    if isinstance(v, Dog)  # which is true even if v is a subclass of Dog
    ```
  – it is regarded a bad practice to write code that explicitly asks about type
    see the article “isinstance considered harmful”:
    “It is easier to ask for forgiveness than permission”
    i.e., try to perform the operation and except the error if it fails,
    instead of cautiously testing first: if ... then ... else ....
    but there are better alternatives yet.
inheritance and class hierarchies

- Java does make a big deal about types (be glad you will be spared it for now!)
  
  ```java
  Mammal m;
  Cat c;
  Dog d;
  m = new Cat();  // legal (implicit cast)
  c = m;          // not allowed by compiler! (it requires a cast)
  c = (Cat)m;     // legal but dangerous!
  m = new Dog();  // legal, but, next, try to cast a Dog into a Cat as follows:
  c = (Cat)m;     // runtime error (even for small dogs and very large cats)
  if (m instanceof Cat)
    c = (Cat)m;
  else if (m instanceof Dog)
    ...
  ```

- Static typing (as in Java) is an attempt to make design decisions explicit
  - preventing coding errors at compile time
  - yet, by permitting casting at run time, bugs can still show up
  - expensive to ask `instanceof` routinely
  - design patterns useful to help reduce need for `instanceof`

- Dynamic typing (as in Python) deemphasizes type manipulation
  - follows a philosophy that casting and is not useful
  - same design patterns will reduce the need for `isinstance`
distributing specializations within a hierarchy

- in a class hierarchy, such as Ducks (and MallardDuck and RubberDuck, ...)
  - variations in behavior with
    - getName()
    - performQuack()
    - fly()
    - display()
    - id()
  - how to implement this variation?
- organize by ‘functional decomposition’?
  - solve all the name issues in one place, the quack issues in another, etc.
- distribute the various methods throughout the hierarchy?
  - each subclass deals with each problem its own way or inherits a method
  - redundant implementations where same behavior (no code reuse)
- the Strategy Pattern
- the Visitor Pattern (an even better solution)
distributing specializations within a hierarchy

the non-OOP way (old-style functional decomposition)

class Duck(object):

    def performQuack(self):
        if isinstance(self, MallardDuck):
            return "quack"
        elif isinstance(self, RubberDuck):
            return "squeek"
        else:
            return "I have no response to that"

    def id(self):
        if isinstance(self, MallardDuck):
            return "I'm a Mallard Duck"
        elif isinstance(self, RubberDuck):
            return "I'm just a Rubber Duck"
        else:
            return "generic"
distributing specializations within a hierarchy

the OOP way (delegate to subclasses and shadow the method in the base class)

```python
class Duck(object):
    def performQuack(self): pass
    def id(self): pass

class MallardDuck(Duck):
    def performQuack(self):
        return "quack"  # needed for other sorts of Duck also!
    def id(self):
        return "I'm a Mallard Duck"

class RubberDuck(Duck):
    def performQuack(self):
        return "squeak"
    def id(self):
        return "I'm just a Rubber Duck"
```
abstract classes in Java

• every **specific** kind of animal has a name
  – but an animal, **in the abstract**, does not.
• every **instance** of an animal is some **specific** type of animal ... so

```java
abstract class Animal {
    abstract public String getName();
}

class Dog extends Animal {
    public String getName() { return "Canis familiaris"; }
}
```

• usage:

```java
Animal a = new Animal();  // won't compile (can't instantiate)
Dog d = new Dog();
```

• **Python has no corresponding notion of an abstract class**
  – no need for it, since variables are not typed
  – no notion of **interfaces** either (which are virtually the same)
    because Python supports multiple inheritance
graphically representing inheritance

Unified Modeling Language, UML
distributing methods within a hierarchy

- so Animal and Mammal illustrate:
  - **superclass**, e.g., Animal is a superclass of Mammal
  - **subclass**, e.g., Mammal is a subclass of Animal
  - Mammal “is derived from” Animal
  - Mammal “extends” Animal

- in general:
  - common methods are put in the highest common superclass
  - methods specific to subclass are in subclass, and abstract in superclass

  Python provides this capability, but not explicitly via **abstract**
  use `pass` instead and shadow the method in superclass
the SimUDuck code from HFDP (Java to Python)

// driver code follows:

public static void main(String[] args) {
    MallardDuck mallard = new MallardDuck();
    RubberDuck rubberDuckie = new RubberDuck();
    DecoyDuck decoy = new DecoyDuck();
    ModelDuck model = new ModelDuck();

    mallard.performQuack();
    rubberDuckie.performQuack();
    decoy.performQuack();

    model.performFly();
    model.setFlyBehavior(new FlyRocketPowered());
    model.performFly();
}

'''driver code follows'''

def main(argv=None):
    mallard = MallardDuck()
    rubberDuckie = RubberDuck()
    decoy = DecoyDuck()
    model = new ModelDuck()

    mallard.performQuack()
    rubberDuckie.performQuack()
    decoy.performQuack()

    model.performFly()
    model.setFlyBehavior(FlyRocketPowered())
    model.performFly()
the SimUDuck code from HFDP (Java to Python)

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {}
    public void setFlyBehavior (FlyBehavior fb) { flyBehavior = fb; }
    public void setQuackBehavior(QuackBehavior qb) { quackBehavior = qb; }
    abstract void display();
    public void performFly() { flyBehavior.fly(); }
    public void performQuack() { quackBehavior.quack(); }
    public void swim() { System.out.println("All ducks float, even decoys!"); }
}

public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```
the SimUDuck code from HFDP (and Java to Python)

```java
public interface QuackBehavior {
    public void quack();
}

public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}

public class Quack extends QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}

abstract public class QuackBehavior {
    abstract public void quack();
}

class QuackBehavior():
    def quack():
        pass

class Quack(QuackBehavior):
    def quack():
        print('Quack')

    def quack():
        print('Quack')
```