ICGE Module 4 Session 2

Object-oriented programming in Python Redux

Topics for today:

- Writing OO classes—a point class
- Other OO Examples:
  - Ising model
  - Iterated prisoners' dilemma
  - Sudoku

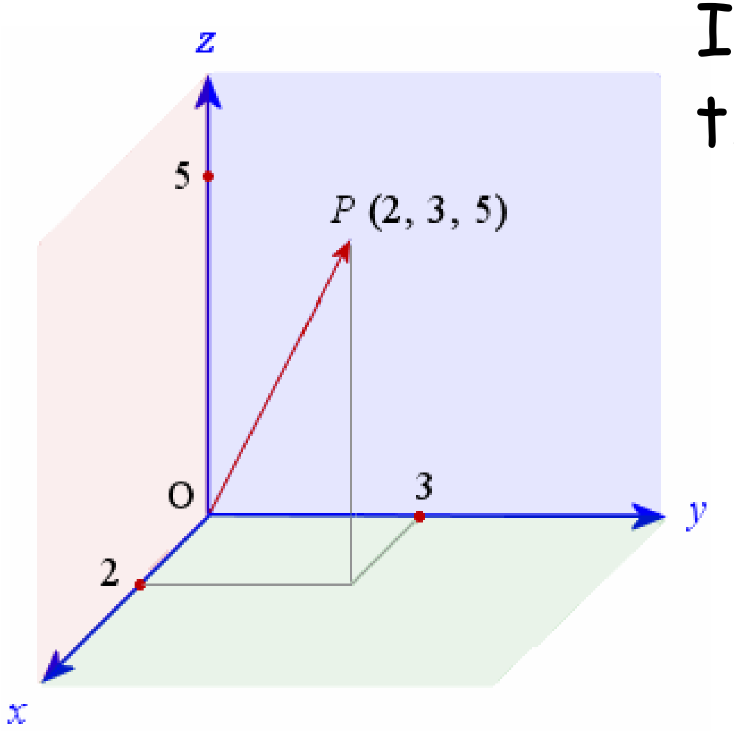
Situations where OO design may not be ideal:

- Performance is a top priority (relevant in OO C++)
- Many developers will be working on the program
- Few obvious "objects" in the task to be programmed

When possible, you should pick the style you like best

Many simulations of physical processes involve vector operations in 3 dimensional space

A 3D point class can simplify codes involving spatial coordinates



In **idle** load and run: **point3d.py** then try these commands:

```
a=point3d(2,3,5)
a.display()
a.sqmag()
b=point3d(5,6,7)
c=a+b
d=5*c
d.display()
d.dist(b)
```

For points (and many useful data types) there are good standard libraries:

NumPy: N-dimensional array "ndarray"
SciPy: More advanced linear algebra on ndarrays

Let's create a simple arbitrary dimensional point class with just a few functions (& no safety net)

Open window and enter the following class and save as **point.R**

```
class point:
    def __init__(self, dim, data):
        self.dim=dim
        self.data=[]
        for i in range(dim):
            self.data.append(float(data[i]))
    def display(self):
        for i in self.data:
            print i,
        print
    def scale(self, x):
        for i in range(self.dim):
            self.data[i]*=x
    def dot(self, a):
        sum=0
        for i in range(self.dim):
            sum+=self.data[i]*a.data[i]
        return sum
```

This is the function that "constructs" new point objects:

`p3=point(2,[3,2])`

`self` is the prefix for data stored in an object

Test your multidimensional point class by writing a short program using the class functions

Be sure to save this in the same folder with **point.py**

```
from point import *
p1=point(4, [1, 4, 5, 2])
p1.display()
p1.scale(3)
p1.display()
p2=point(4, [5, 1, 2, 3])
print "p1 dot p2=", p1.dot(p2)
p3=point(2, [3,2])
p3.display()
print "p3 dot p2=", p3.dot(p2)
```

Same operation in a procedural code would require a few lines but may run much faster:

```
float dot=0.;
for (int i=0; i<dim; i++) {
      dot+=p1[i]*p2[i];
}
```

Or much, much faster*

```
r1 = _mm_mul_ps(p1, p2);
r2 = _mm_hadd_ps(r1, r1);
r3 = _mm_hadd_ps(r2, r2);
_mm_store_ss(&dot, r3);
```

*SSE calls for dim=4

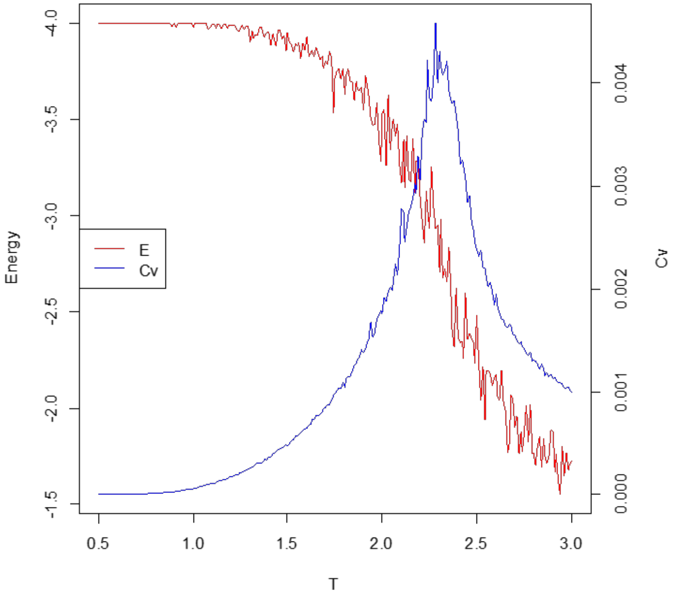
This is an “unsafe” class since it will try to execute bad operations (like the dot product between vectors of different length)

“Ising models” are very simple spin lattices that undergo fairly realistic “phase transitions”

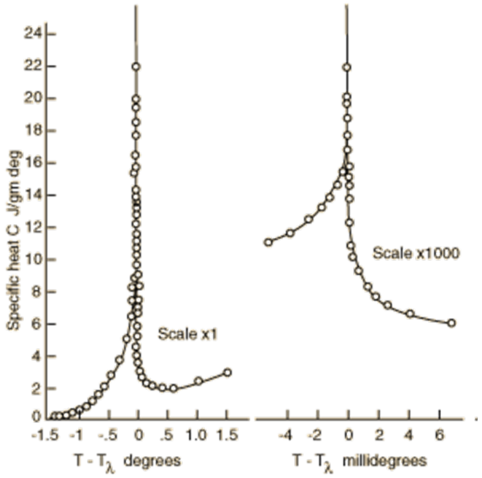
6X6 Ising model

↓	↑	↓	↓	↑	↑
↑	↑	↑	↓	↑	↓
↑	↑	↓	↓	↓	↓
↓	↑	↓	↓	↑	↓
↑	↑	↑	↓	↑	↓
↓	↓	↓	↓	↑	↑

Ising model melting at T=2.3



Helium fluid→Superfluid transition (@2.17K)

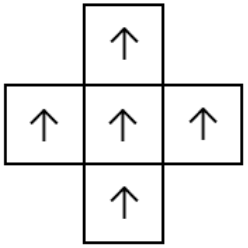


Graph from:
http://hyperphysics.phy-astr.gsu.edu/hbase/lhel.html

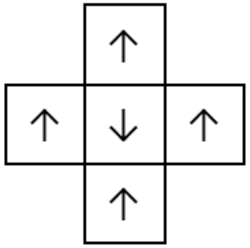
$\sigma(\uparrow)=+1$

$\sigma(\downarrow)=-1$

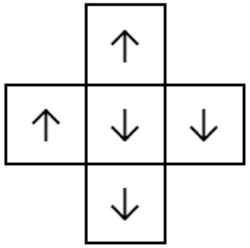
$$E_i = -\sum_j \sigma_i \, \sigma_j$$



Best:
Energy of central spin is -4



Worst:
Energy of central spin is +4



In between:
Energy of central spin is 0

Functions in class library `ising_class.py` for running & analyzing 2-dimensional Ising models

Function	Function name and args, example of use
Create an ising model with a specified temperature, n (spins on one side)	`ising(temp, n)` `ising1=ising(2.4, 10)`
Print out the ising system to the screen	`printsys()` `ising1.printsys()`
Run a single trial (flip 1 spin)	`trial()` `ising1.trial()`
Run multiple trials (flip m spins)	`trials(m)` `ising1.trials(100000)`
Set the system temperature to a new value	`changeTemp(newtemp)` `ising1.changeTemp(3.4)`
Randomize the spins (equal prob up or down)	`randomize()` `ising1.randomize()`
Reset sums for calculation energy and magnetization statistics	`resetprops()` `ising1.resetprops()`
Calculate energy and magnetization for current state of system and add to running sums	`addprops()` `ising1.addprops()`
Calculate and print out system properties	`calcprops()` `ising1.calcprops()`

The class library makes it easy to assemble Ising simulations where all details are hidden

---

Load into **idle** the program **ising1.py**

```
from ising_class import *
ising1=ising(2.3, 20)
ising1.printsys()
ising1.resetprops()
ising1.randomize()
ising1.trials(5000)
ising1.resetprops()
for i in range(50000):
    ising1.trial()
    ising1.addprops()
ising1.calcprops()
ising1.printsys()
```

Numbers output by calcprops()

T $\langle E\rangle$ σ_E $\langle M\rangle$ σ_M

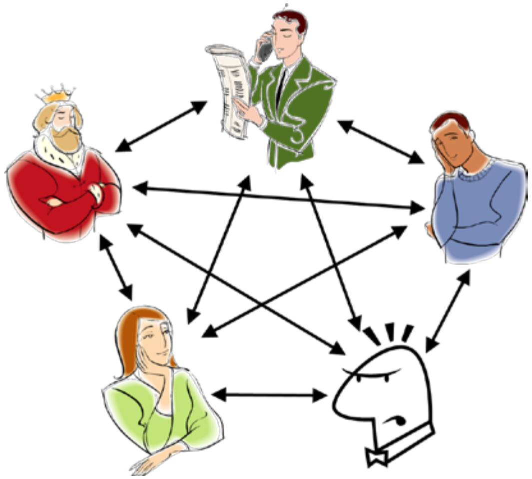
```
2.3000 -3.1472 0.0021 0.0175 0.0012
```

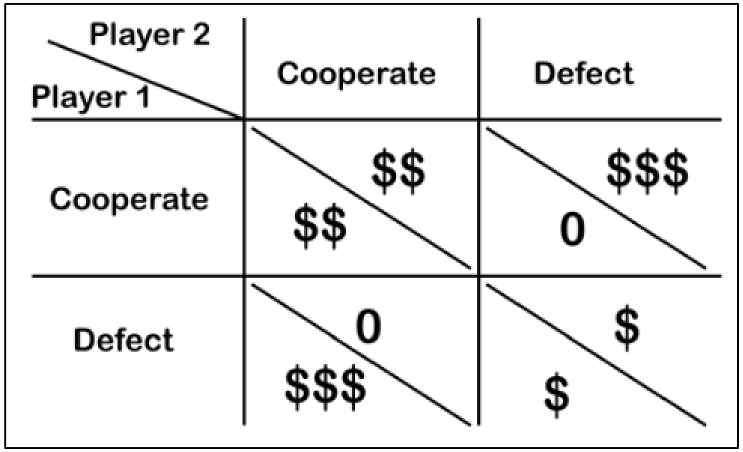
These diverge at the "melting" temperature

For a program that scans temperature to find melting temperature, see posted **ising2.py**

The Iterated Prisoner's Dilemma (IPD) is a simple model for repeated business or social interactions

Multiple players repeatedly have pairwise transactions, deciding to "Cooperate" or "Defect" each time:





Player 1 \ Player 2	Cooperate	Defect
Cooperate	$$ $$	$$$ 0
Defect	0 $$$	$ $

"Friendly" Transactions

Player 1	Player 2	Player 1 Total	Player 2 Total
Cooperate	Cooperate	2	2
Cooperate	Cooperate	4	4
Cooperate	Cooperate	6	6
Cooperate	Defect	6	9

"Hostile" Transactions

Player 1	Player 2	Player 1 Total	Player 2 Total
Cooperate	Defect	0	3
Defect	Defect	1	4
Defect	Defect	2	5
Defect	Defect	3	6

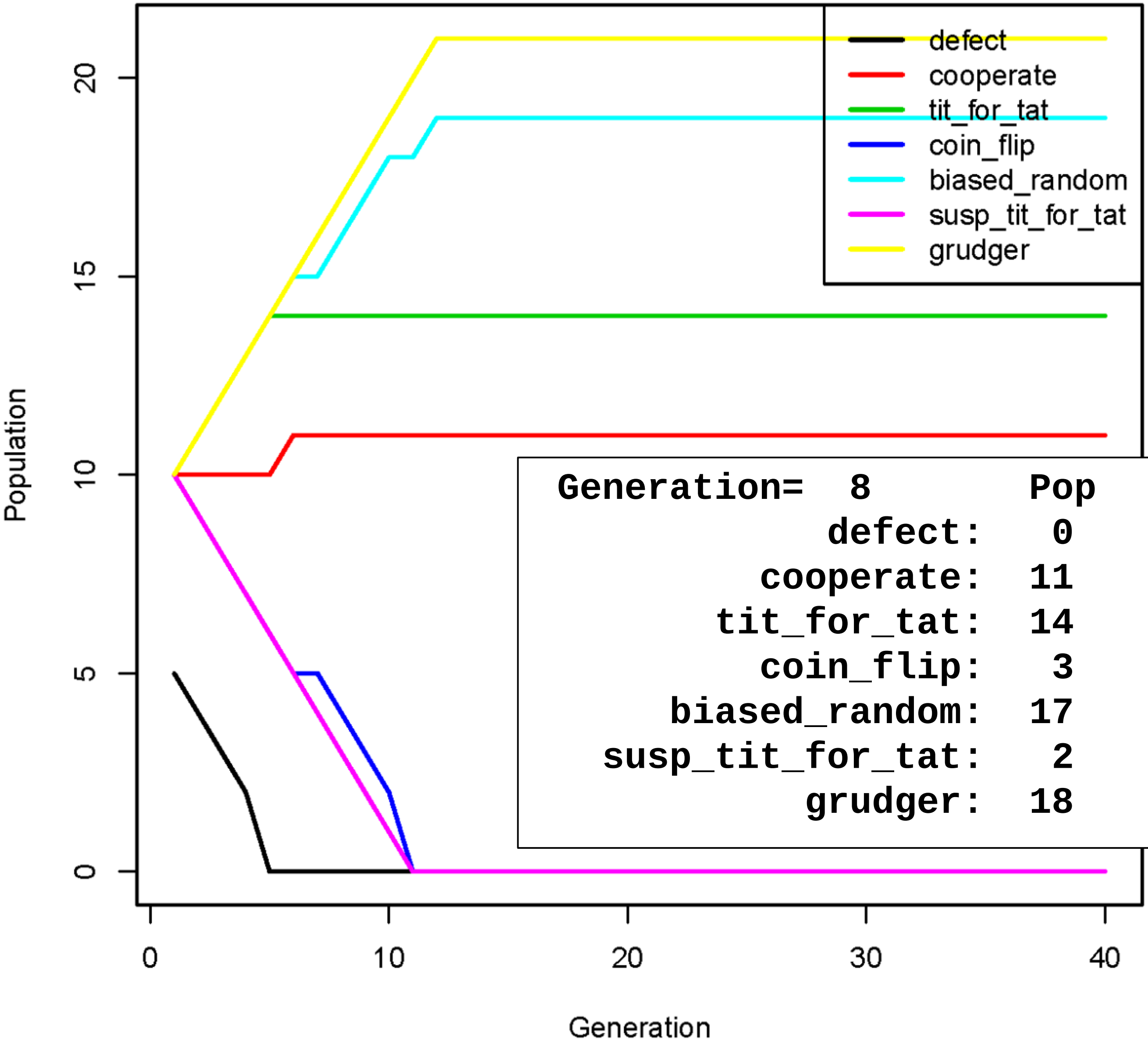
In the early 1980's Robert Axelrod at Michigan ran a series of IPD "tournaments"

Examples of some simple IPD strategies

Name	Strategy
Always Cooperate	Always cooperate
Always Defect	Always defect
Tit for Tat	Cooperate first, and then do what opponent did last time
Suspicious Tit for Tat	Defect first, and then do what opponent did last time
Coin flip	Defect or cooperate with equal probability
Biased Random	Defect or cooperate with prob. biased by opponent's history
Grudger	Cooperate until opponent defects, then always defect

Best deterministic strategy in Axelrod's study

The IPD can be put in a simulation of Darwinian evolution where species fitness = average score



Load & run: **`ipd.py`**
uses **`ipd_class.py`**

```
Generation=  8        Pop        Score       New Pop
            defect:    0        0.0000         0
         cooperate:   11       21.4295        11
       tit_for_tat:   14       27.4481        14
         coin_flip:    3        5.0558         2
     biased_random:   17       33.1816        18
  susp_tit_for_tat:    2        3.1673         1
           grudger:   18       35.4279        19
```

The evolutionary IPD simulation program **ipd.py** allows setting the initial populations

You set the initial composition of the environment on these lines:

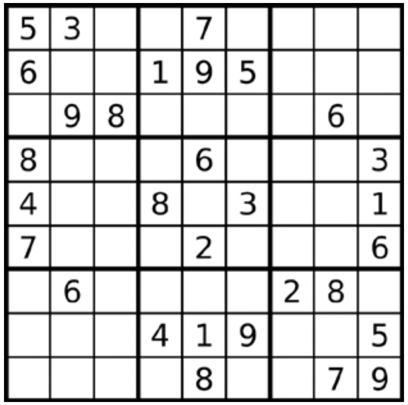
```
### Strategies available ###
strats=[defect,cooperate,tit_for_tat,coin_flip,
        biased_random,susp_tit_for_tat,grudger]

### Set list for the number of each strategy ###
Nactor_list=[5, 15, 20, 10, 10, 10, 10]
```

You can also add new strategies by adding new player classes

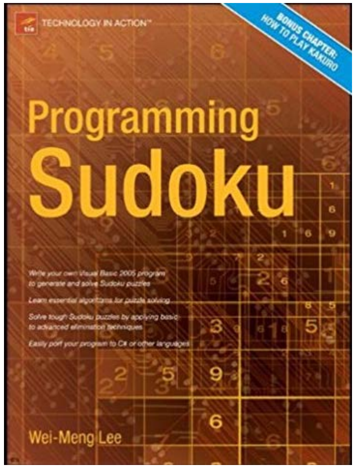
```
class waffler:
    def __init__(self,Nactors,myid):
        self.Nactors=Nactors
        self.myid=myid
        self.name="waffler"
        self.responses=["Cooperate","Defect"]
        self.next=1
    def response(self, other):
        self.next=(self.next+1)%2
        return self.responses[self.next]
    def inform(self, other, other_response):
        return
```

Example of OO encapsulation: Sudoku--a simple, but for many very addictive, numerical puzzle



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Goal: Fill in digits 1-9 so that there are no repeated digits in any row, column or 3x3 sub-block



Load and run **`sudoku_class.py`**

`s=sudoku()` Create an empty 9x9 Sudoku grid

`s.makepuzzle(36)` Fill in 36 number clues (or any # < 81)

`s.display()` Print out current Sudoku grid

`s.solve()` Try to solve the puzzle (without using any guesses)

`s.solved()` Is the puzzle completely solved?

`s.generate()` Generate a completely solved Sudoku puzzle

Program to calc. solve rate vs # clues: **`sudoku.py`**