ICGE Programming Module--Python!

Part 2: Object-oriented programming in Python

Imagine you want to simulate something:

What will your program need to include?

- Variables to store the properties of each component (cards, frogs, etc.)
- Logic and math to change these variables (deal card, move frog, etc.)
- Steps to initialize and print out the properties of each component

What's the best way to organize these different pieces?

"Object-oriented" programming organizes your program around the natural "objects" involved

Frog "Object"

Data:
- gender
- age
- health
- hunger

Set age
Feed
Get hunger level

Functions to operate on object

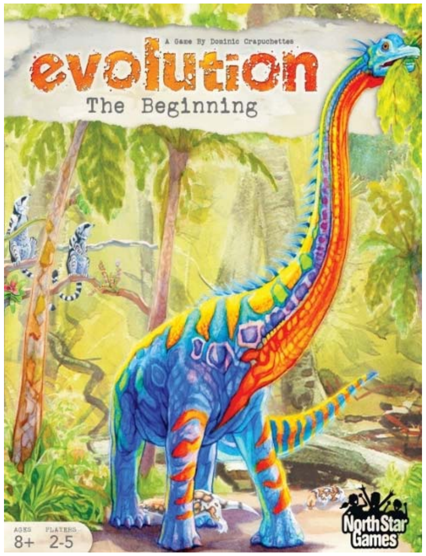
Deck "Object"

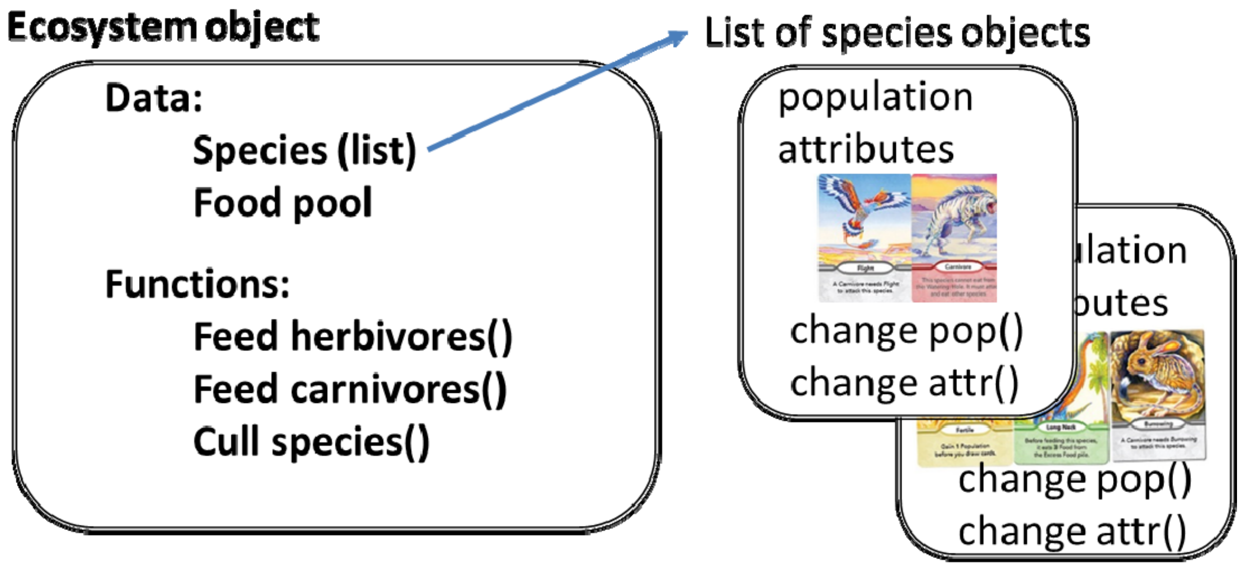
Data:
- List of cards
- #cards

Shuffle deck
Get #cards left
Deal a card

Functions to operate on object

"OO" programming is an intuitive & fun approach to designing many types of simulation programs





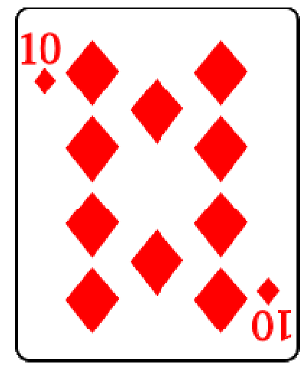
Promised advantages of OO programming

- Simplifies programming by hiding the details of each component of the program
- Improved reliability since each class can be independently debugged
- Improved code reuse and sharing since you only need to remember the class "interface" and don't need to know the details of how the code is implemented

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

Let's try out two simple classes that implement a deck of playing cards and an individual card



Deck object		Card object	
Create deck	__init__()	Create card	__init__()
Shuffle deck	shuffle()	What type of card?	type()
Look at whole deck	printdeck()	What suit?	suit()
Deal a card	dealcard()	What is the card value? (depends on card game)	value()
How many cards left?	cardsleft()	Look at card	printcard()

Start idle, then open and run the file **cards.py**

Create a deck object and try some of its functions:

```
adeck=deck()
adeck.shuffle()
adeck.printdeck()
for i in range(15):
  acard=adeck.dealcard()
  print "acard:",acard.printcard()
print "# left:",adeck.cardsleft()
adeck.shuffle()
adeck.printdeck()
bdeck=deck()
bdeck.printdeck()
```

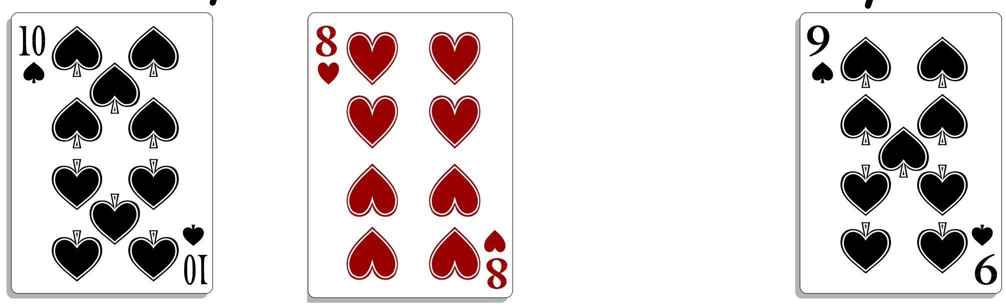
Let's use this card "class" to build a simple card game and determine players' odds of winning

Rules:

1. Player A gets 2 cards & Player B gets 1 card
2. Player A wins the hand if either card has a greater value than Player B's card
3. Play though entire deck and tally hands won

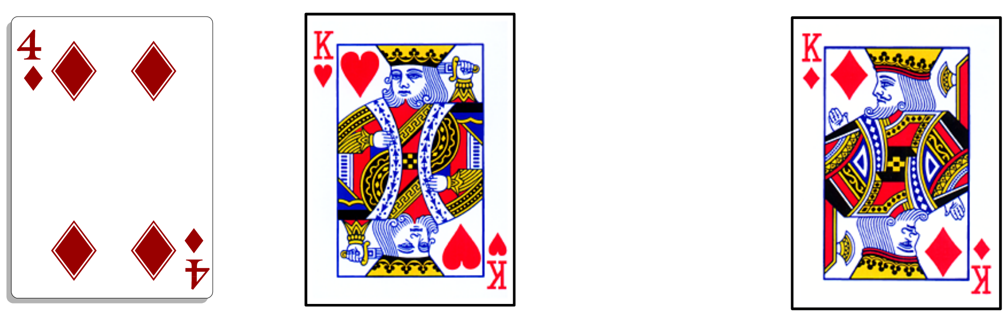
Player A | Player B

Hand 1:



A wins

Hand 2:



B wins

Here's a program that plays 10000 trials of this game and prints the final win statistics

Enter the following and save in the same directory with the file **cards.py**

```
from __future__ import division
from cards import *
ntrials=10000
awins=0
for i in range(ntrials):
    adeck=deck()
    adeck.shuffle()
    ascore=0
    bscore=0
    while adeck.cardsleft()>2:
        acard1=adeck.dealcard()
        acard2=adeck.dealcard()
        bcard=adeck.dealcard()
        if acard1.value()>bcard.value() or acard2.value()>bcard.value():
            ascore+=1
        else:
            bscore+=1
    if ascore > bscore:
        awins+=1
print("Player A win percentage=",awins/ntrials)
```

The card values are set in the **deck class** and can be changed by editing the numerical values

Load **cards.py** into **idle** and look for following lines:

```
class deck:
    def __init__(self):
        self.deck=[]
        suits=['S','C','H','D']
        values={'A':1,'2':2,'3':3,'4':4,'5':5,'6':6,'7':7,'8':8,'9':
9,'10':10,'J':10,'Q':10,'K':10}
        types=['A','2','3','4','5','6','7','8','9','10','J','Q','K']
```

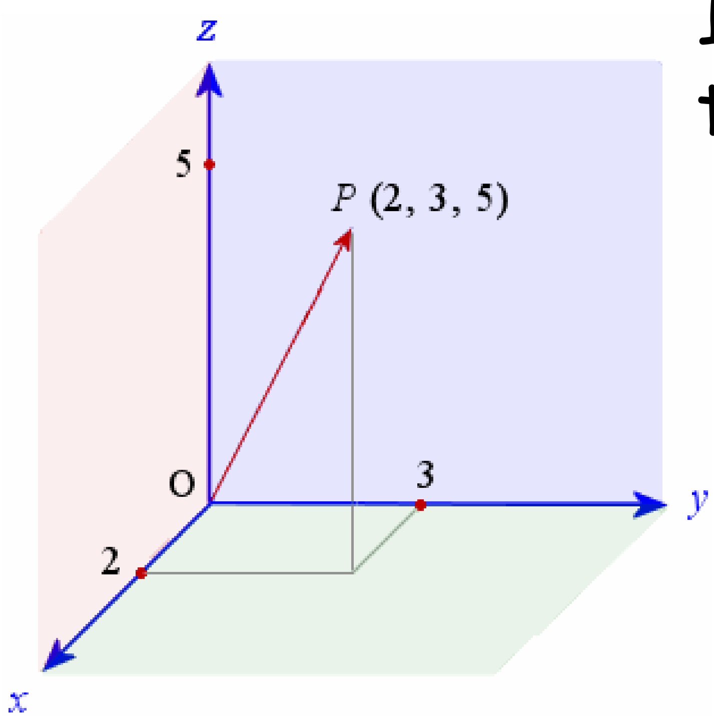
Player B wins when cards are equal, so giving more cards equal values will help this player. Edit the **cards.py** file and make this change (save your changes before rerunning **gameMC.py**)

```
values={'A':1,'2':2,'3':3,'4':4,'5':5,'6':6,'7':7,'8':9,'9':9
,'10':10,'J':10,'Q':10,'K':10}
```

The most balanced version of the program I could find gave Player A a 50.5% chance of winning—can you do better?

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.py**

```
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*

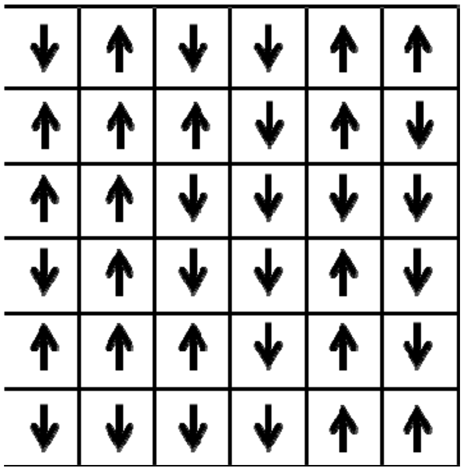
```
r1 = _mm_mul_ps(p1, p2);
r2 = _mm_hadd_ps(r1, r1);
r3 = _mm_hadd_ps(r2, r2);
_mm_store_ss(&dot, r3);
```

*SSE calls for dim=4

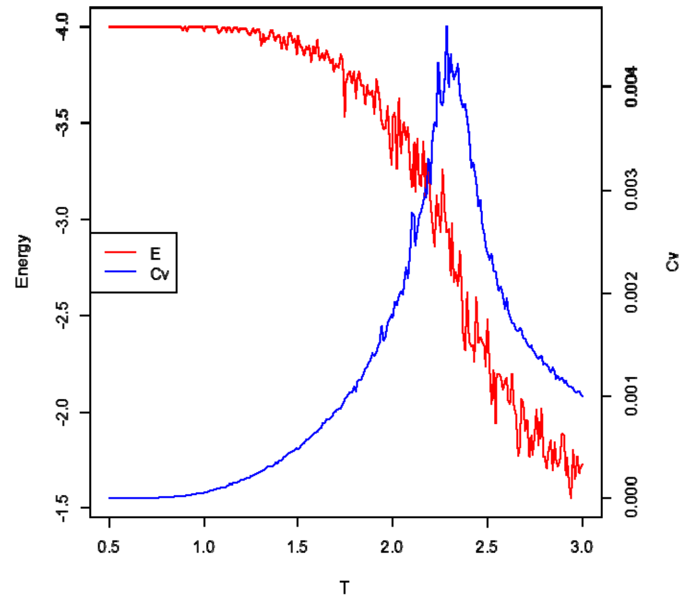
This is an "unsafe" class since it will try to execute bad operations (like the dot product between vectors of different length)

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

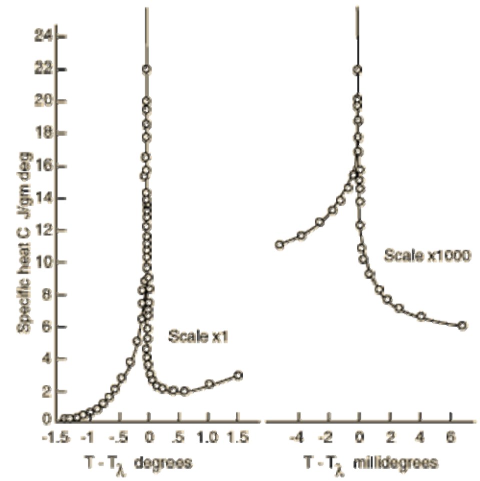
6X6 Ising model



Ising model melting at T=2.3



Helium fluid→Superfluid transition (@2.17K)

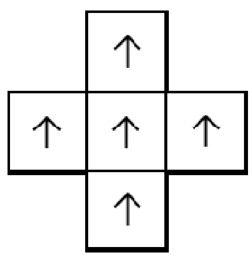


Graph from:
http://hyperphysics.phy-astr.gsu.edu/hbase/lhel.html

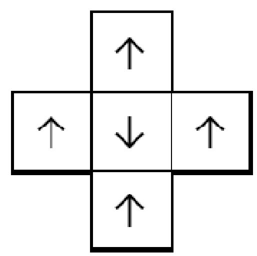
$\sigma(\uparrow)=+1$

$\sigma(\downarrow)=-1$

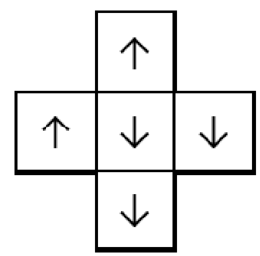
$$E_i = -\sum_j \sigma_i \sigma_j$$



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()

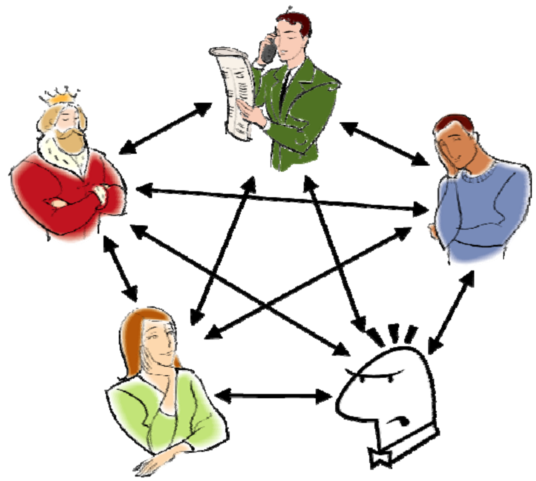
T	$\langle E \rangle$	σ_E	$\langle M \rangle$	σ_M
2.3000	-3.1472	0.0021	0.0175	0.0012

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

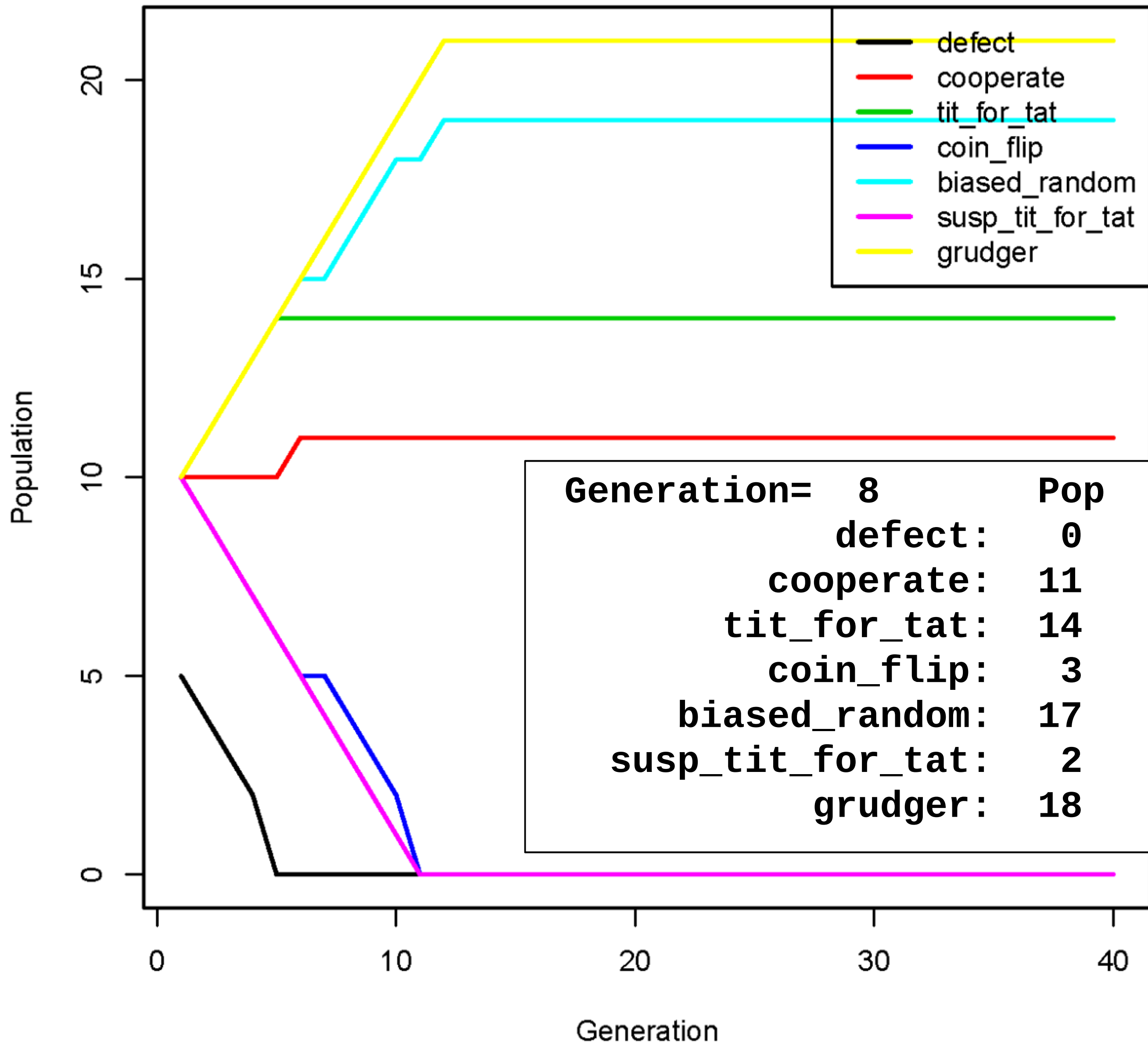
In the early 1980's Robert Axelrod at Michigan ran a series of multi-player IPD "tournaments"

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



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

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

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]
```

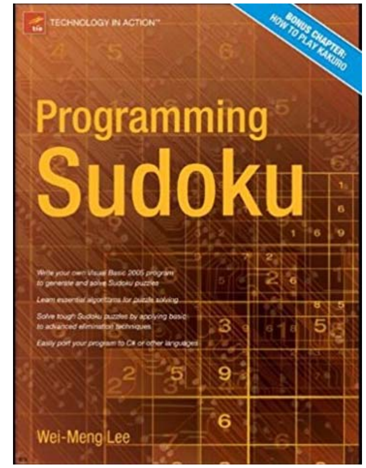
You can also add new strategies by adding new player classes

```
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

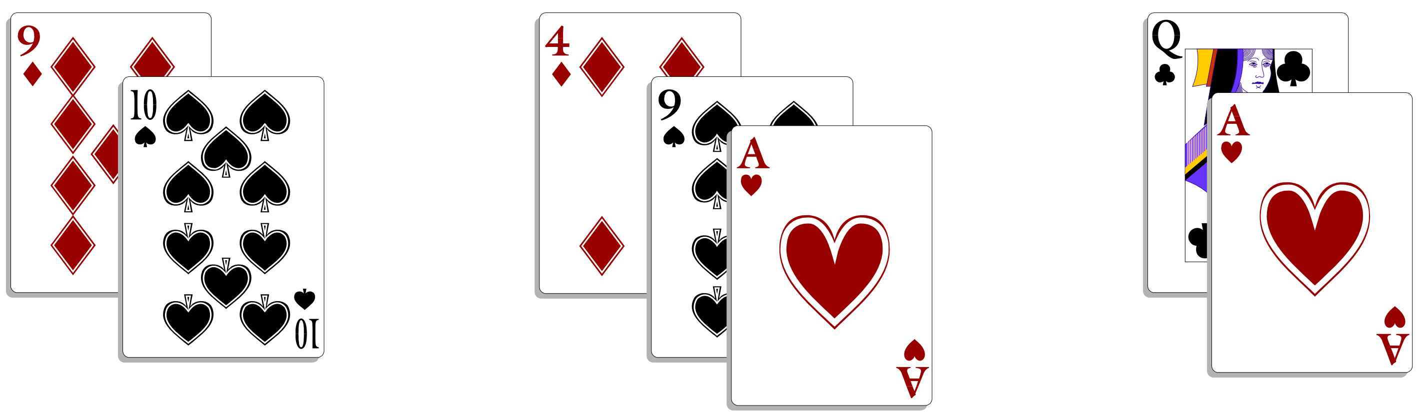
Program to calc. solve rate vs # clues: **sudoku.py**

Blackjack is a slightly more complex game where winning depends on the point value each hand

Goal: Get a set of cards totaling as close as possible to 21, without going over 21

Card values:

- 2, 3, 4, 5, 6, 7, 8, 10: Value of number
- J, Q, K: Count as 10
- A: Count as 1 or 11



Rules of blackjack (simplified)

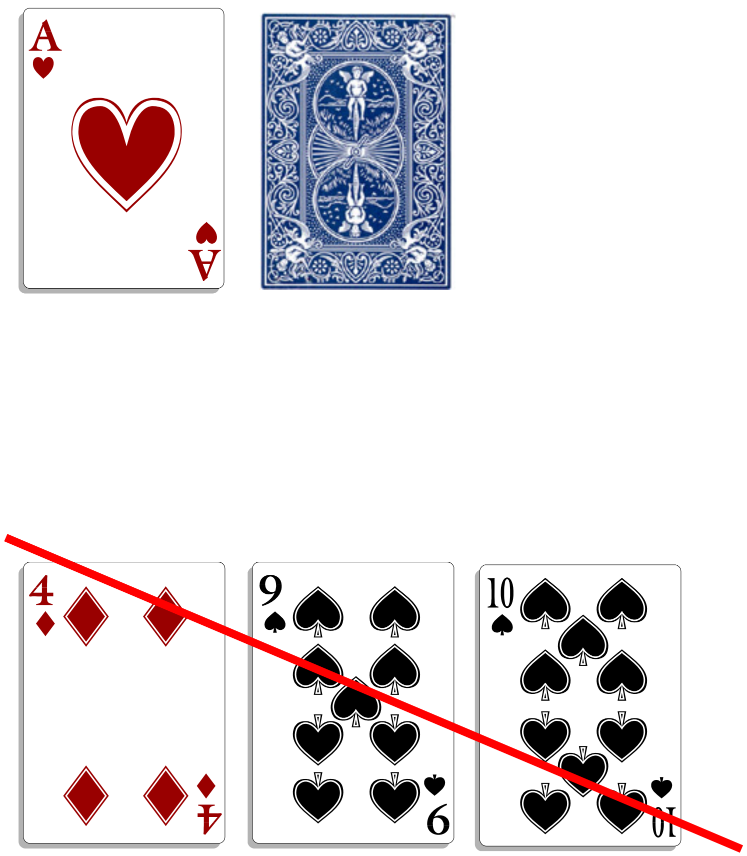
Players: 1 player and 1 dealer

Rules:

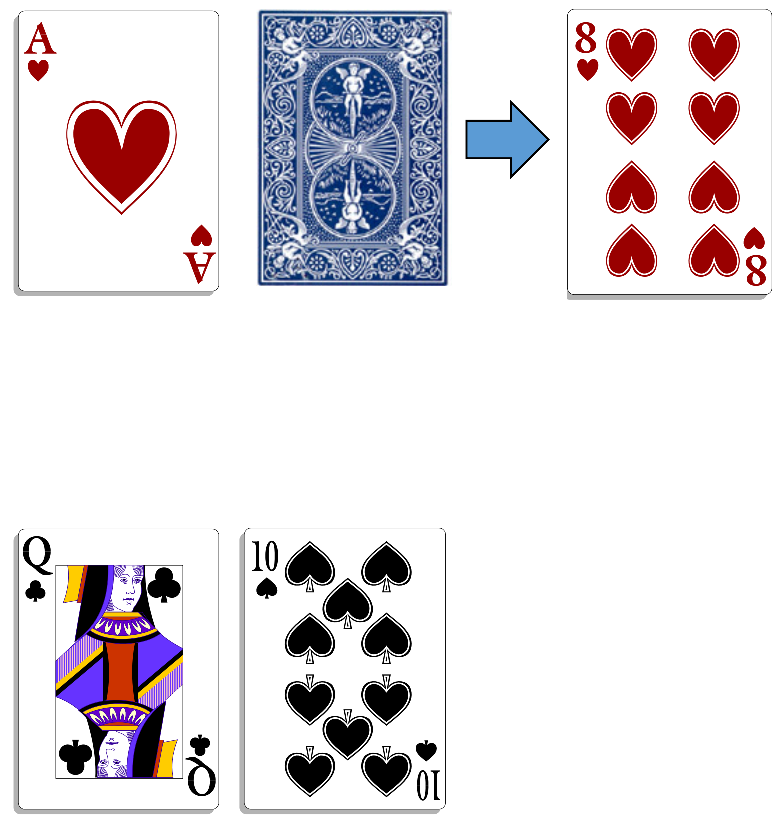
- Deal two cards to player & dealer with one of the dealer's cards face up
- Player goes first, requesting as many cards as he wants ("hits")
- If player goes over 21, he "busts" and dealer wins
- If player doesn't bust, dealer takes cards up to a cutoff of 17 or a bust
- Player & dealer compare scores; dealer wins in a tie

Two sample hands of Blackjack

DEALER



Player Busted !



Player wins !

PLAYER

You can change the player's strategy and use Monte Carlo to test effectiveness

Things to change in strategy:

- Player's cutoff to take new card (recalling that dealer must "hold" at 17)
- How to use information about what cards the dealer is showing—Typically the higher the card the dealer is showing, more likely you will benefit by taking another card

	Dealer's Up Card									
Your Hand	2	3	4	5	6	7	8	9	10	A
8	H	H	H	H	H	H	H	H	H	H
9	H	D	D	D	D	H	H	H	H	H
10	D	D	D	D	D	D	D	D	H	H
11	D	D	D	D	D	D	D	D	D	H
12	H	H	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
16	S	S	S	S	S	H	H	H	H	H
17	S	S	S	S	S	S	S	S	S	S

http://www.hitorstand.net/strategy.php

Program **blackjack.py** on CatCourses is a Monte Carlo simulation of the game

The program plays 10000 games of blackjack following the specified player strategy

Output:

```
>>>
Ntrials= 10000
Player wins:  4244
Dealer wins:  5756
Player wins:  42.44 percent
```

The player strategy can be modified by editing the **holdlimit** variable in the **playerclass**

```
################ Define Python classes for simulation ######################
class playerclass:
    def __init__(self):
        self.holdlimit={'A':17,'2':17,'3':17,'4':17,'5':17,'6':17,'7':17,\
                        '8':17,'9':17,'10':17,'J':17,'Q':17,'K':17}
```

You specify the player's strategy in terms of the hold value under different conditions

Code: **blackjack.py**

Dealer's exposed card

Player's hold limit for that showing card

```
class playerclass:
     def __init__(self):
         self.holdlimit={'A':17,'2':17,'3':17,'4':17,'5':17,'6':17,'7':17,\
                         '8':17,'9':17,'10':17,'J':17,'Q':17,'K':17}
```

Example: hold limit of 17 in all cases

Example: variable hold limit

```
class playerclass:
     def __init__(self):
         self.holdlimit={'A':17,'2':12,'3':13,'4':14,'5':15,'6':16,'7':17,\
                         '8':17,'9':17,'10':17,'J':17,'Q':17,'K':17}
```