```
1 from pylab import *
```

English Auctions

English auctions

English auctions are the kind of auctions we are most familiar with.

- an item is shown
- the auctioneer solitics bids from all participants
- participants can choose to make a bid higher by some increment or can choose not to bid
- the highest bid wins

Dropping out vs Not Bidding

Rational agents in a setting with no adaptation or interaction would simply bid until they reach some limit.

Not bidding when polled bears the risk that the auction is won by someone else (opportunity cost).

However, agents might sit out one round if they assume that other agents are observing their actions, or because the last bids caused them to conclude that the value of the item is lower, but they revise that estimate in light of later bids.

Private Value Model

How is the value of the item determined?

- We assume that there is a base value normally distributed according to $V \sim G(\mu, \sigma)$
- There is an additional (optional) private value variation with distribution $V_p = V + Q$ where $Q \sim G(0, \sigma_p)$
- Each participant receives an estimate of their private value $\tilde{V}_p = V_p + \nu$ where $\nu \sim G(0, \sigma_u)$

Information sources

There are two source of information about the value of an item:

- the private estimate $\tilde{V}_p$
- the collection of bids made by other participants in the auction

Profit

The profit of every participant over time is the difference of:

- the private value gained
- the money paid in bids

auctioneer for an English auction

```
class Auctioneer:
    def __init__(self,participants,mu=100.0,sigma=20.0,psigma=0.0,
        uncertainty=10.0):
        self.participants = participants
        self.mu = mu
        self.sigma = sigma
        self.psigma = psigma
        self.uncertainty = uncertainty
        self.increment = 1.0
    def genvalue(self):
        return clip(self.mu+randn()*self.sigma,0.0,1e6)
    def auction(self,verbose=0):
        value = self.genvalue()
        for p in self.participants:
            pvalue = clip(value + randn() * self.psigma,0.0,1e6)
            estimate = pvalue + randn() * self.uncertainty
            p.pvalue = pvalue
            p.giveEstimate(estimate)
        last = -self.increment
        winner = None
        for p in self.participants:
            p.startBidding(len(self.participants))
        for round in range(1000):
            success = 0
```

The simple bidder just bids up to a fraction of their estimate of the private value of the item.

simple bidder

```
class Bidder:
    def __init__(self,maxbid=0.8):
        self.balance = 0.0
        self.value = 0.0
        self.maxbid = maxbid
    def giveEstimate(self,estimate):
        self.estimate = estimate
    def startBidding(self,n):
        self.bidders = [-1.0]*n
    def announce(self,bid,bidder):
        self.bidders[bidder] = bid
    def solicitBid(self,last,incr):
        if last+incr<self.maxbid*self.estimate:
            return last+incr
    def gain(self):
        return self.value+self.balance
```

```
participants = [Bidder() for i in range(5)]
auctioneer = Auctioneer(participants)
auctioneer.gains()
```

```
(2.205314868771302,
 1.2098102298952653,
 2.0889592570658557,
 2.263475877601504,
 2.486365849422504 5)
```

Winner's Curse

In a symmetric setting, `maxbid` for this choice of distributions should be set somewhat below the true value of the item, otherwise the winner is afflicted by the *winner's curse*. That's because the auction select for people who accidentally overestimate the private value of the item.

```
for m in linspace(0.5,1.0,6):
    participants = [Bidder(maxbid=m) for i in range(5)]
    auctioneer = Auctioneer(participants)
    print m,mean(auctioneer.gains(N=1000))
```

```
0.5 8.9460120351
0.6 6.65084953128
0.7 4.49585360683
0.8 2.17616941795
0.9 0.0209089858165
1.0 -2.16911006642
```

Using Information of Other Participants

A smart bidder uses information from other participants in the auction. Let's create a simple smart bidder

- the bidder has a safe limit and a risky limit
- he stays in the auction if below the safe limit, or if below the risky limit and there are still enough other participants to make it seem safe

```
class SmartBidder:
    def __init__(self,maxbid=0.8,risky=1.0):
        self.balance = 0.0
        self.value = 0.0
        self.maxbid = maxbid
        self.risky = risky
    def giveEstimate(self,estimate):
        self.estimate = estimate
    def startBidding(self,n):
        self.bidders = [0.0]*n
        self.active = n
    def announce(self,bid,bidder):
        if bid is None and self.bidders[bidder] is not None:
            self.active -= 1
        self.bidders[bidder] = bid
    def solicitBid(self,last,incr):
        if last+incr<self.maxbid*self.estimate:
            return last+incr
        if last+incr<self.risky*self.estimate and self.active>=3:
            return last+incr
        return None
    def gain(self):
        return self.value+self.balance
```

```
for m in linspace(0.5,1.0,6):
    participants = [SmartBidder(maxbid=m,risky=m+0.2)]+[Bidder(
        maxbid=m) for i in range(4)]
    auctioneer = Auctioneer(participants)
    g = auctioneer.gains(N=10000)
    participants = [Bidder(maxbid=m+0.2)]+[Bidder(maxbid=m) for i
        in range(4)]
    auctioneer = Auctioneer(participants)
    h = auctioneer.gains(N=10000)
    print m,"smart",g[0],"risky",h[0],"smart-others",mean(g[1:]),"
        risky-others",mean(h[1:])
```

```
0.5 smart 15.0150922895 risky 28.624387843 smart-others 7.38789361547 risky-others
0.6 smart 10.3083580541 risky 18.1339082998 smart-others 5.82506507554 risky-others
0.7 smart 6.8578690038 risky 7.8654160059 smart-others 3.86668288789 risky-others 0
0.8 smart 3.31861787529 risky -1.62269257178 smart-others 1.95440353669 risky-other
0.9 smart 0.258843172478 risky -10.3465441388 smart-others -0.0890251015037 risky-o
1.0 smart -2.6061725043 risky -18.0559950088 smart-others -2.14888575995 risky-othe
```

Population Dependency of Strategies

Note that a smart bidder beats a risky bidder vs regular bidders.

```
for m in linspace(0.5,1.0,6):
    participants = [SmartBidder(maxbid=m,risky=m+0.2) for i in
        range(5)]
    auctioneer = Auctioneer(participants)
    g = auctioneer.gains(N=10000)
    participants = [Bidder(maxbid=m) for i in range(5)]
    auctioneer = Auctioneer(participants)
    h = auctioneer.gains(N=10000)
    print m,"smart",mean(g),"dumb",mean(h)
```

```
0.5 smart 5.88616474396 dumb 8.91795009093
0.6 smart 3.90990204248 dumb 6.69898707353
0.7 smart 1.89614093175 dumb 4.47528150276
0.8 smart -0.136330204419 dumb 2.23436647775
0.9 smart -2.18623821023 dumb 0.00722432760355
1.0 smart -4.23641026388 dumb -2.2471004348
```

When populations of smart bidders are compared with populations of simple bidders, we find

- smart bidders do worse than simple bidders in that context
- performance therefore does not just depend on the individual strategy, but the collection of strategies
- smart bidders do better for the *seller* in this context

What strategy can you adopt to influence smart bidders?

Shills

- Sellers of items secretly bid on their own items to above their expected value.
- Bidders see the price signal from these shills and bid higher than they otherwise would.
- If all bidders drop out, the seller buys his own item and has no loss (or a small loss from transaction costs).

Questions / Exercises

Write a smart `Bidder` that wins as much as possible against other dumb and smart `Bidders`.

Put your bidder implementation into a source file `yourname{\it bidder.py` and call it `class Bidder`. We will run a tournament and import `yourname}bidder.Bidder()`.