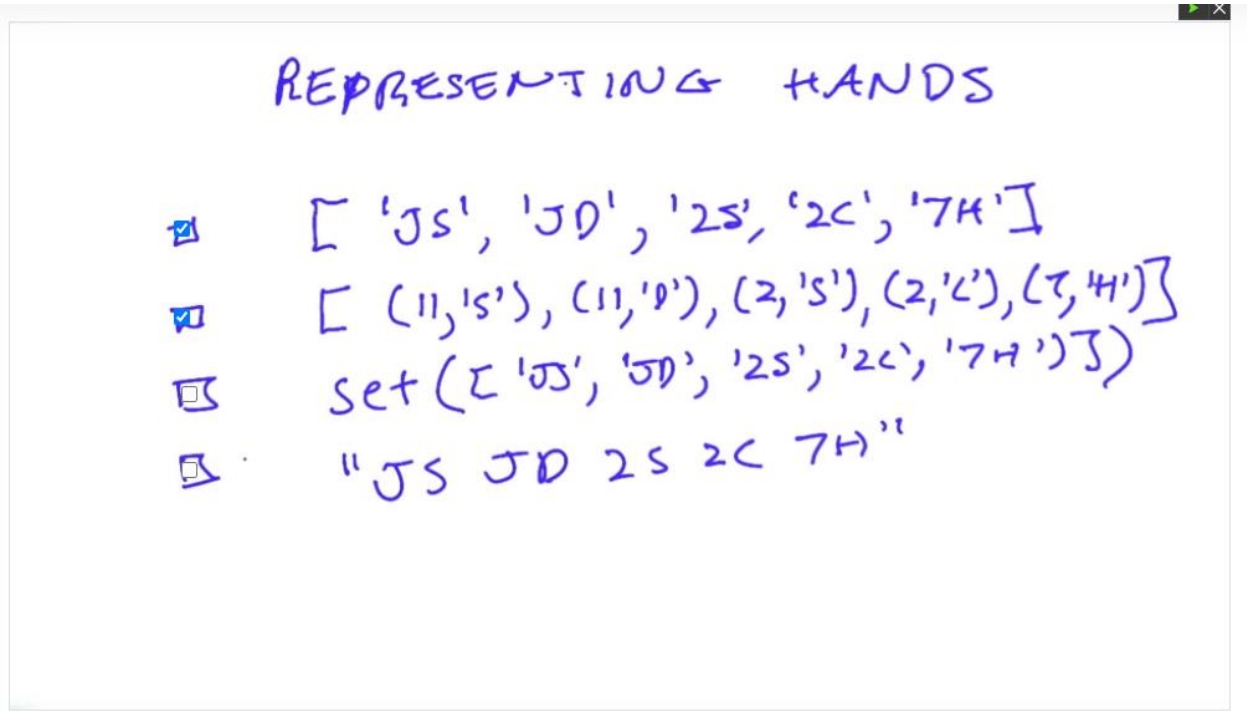
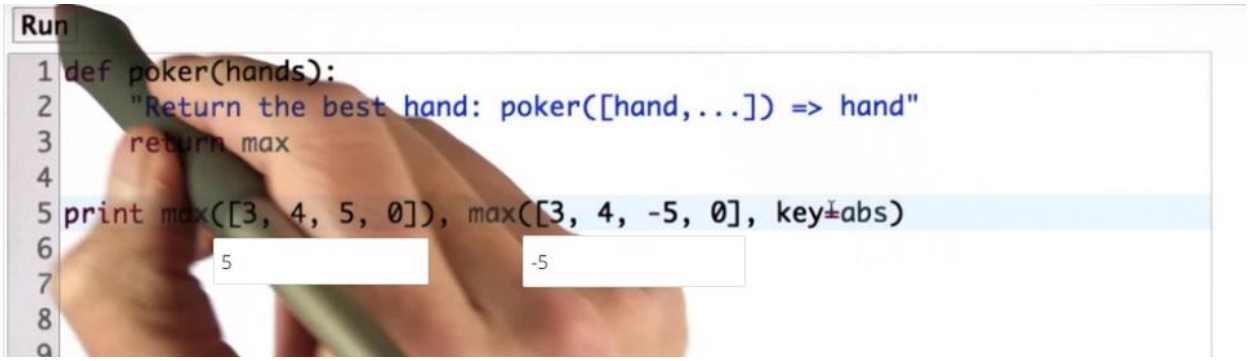
Fellhipe Gutierrez

Representing Hands



Poker Function

Understanding Max



Using Max

```
def poker(hands):
    "Return the best hand: poker([hand,...]) => hand"
    return max(hands,key=hand_rank)
```

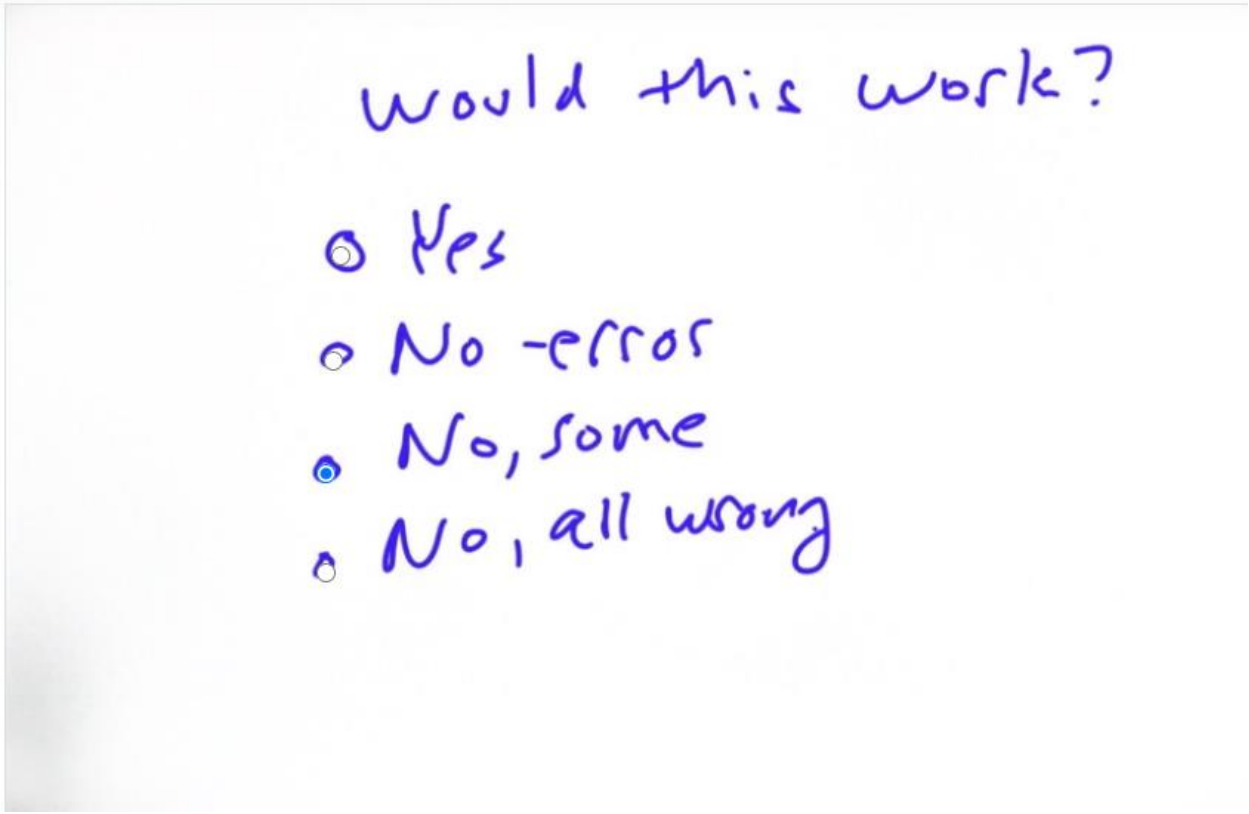
Testing

```
def test():
    "Test cases for the functions in poker program"
    sf = "6C 7C 8C 9C TC".split() # => ['6C', '7C', '8(
    fk = "9D 9H 9S 9C 7D".split()
    fh = "TD TC TH 7C 7D".split()
    assert poker([sf, fk, fh]) == sf
    assert poker([fk,fh])==fk
    assert poker([fh,fh])==fh
    return "test pass"
```

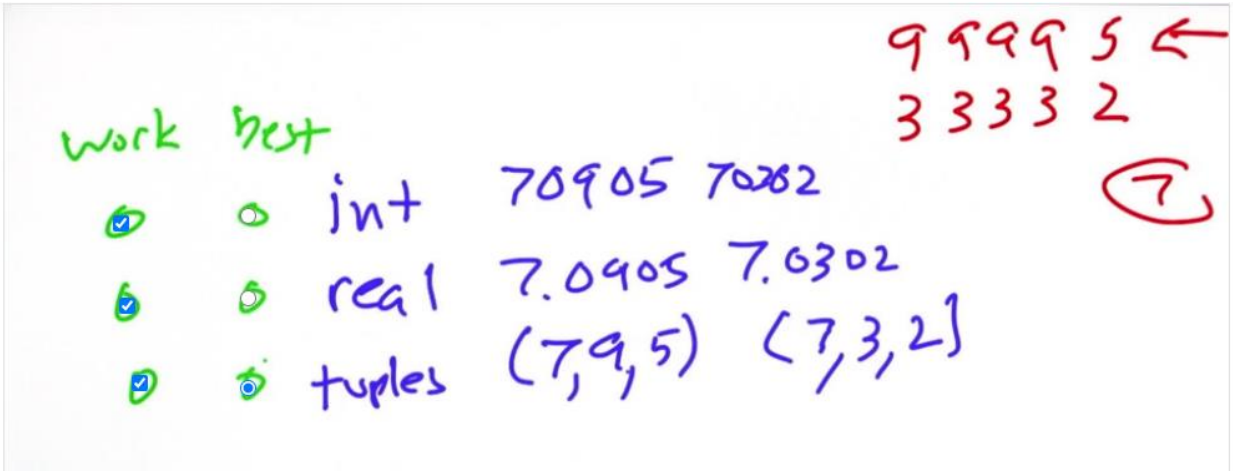
Extreme Values

```
def test():
    "Test cases for the functions in poker program"
    sf = "6C 7C 8C 9C TC".split()
    fk = "9D 9H 9S 9C 7D".split()
    fh = "TD TC TH 7C 7D".split()
    assert poker([sf, fk, fh]) == sf
    assert poker([fk, fh]) == fk
    assert poker([fh, fh]) == fh
    assert poker([sf]) == sf
    assert poker([sf]+99*[fh]) == sf
    return "test pass"
```

Hand Rank Attempt



Representing Rank



Testing Hand Rank

```
def test():
    "Test cases for the functions in poker program"
    sf = "6C 7C 8C 9C TC".split() # Straight Flush
    fk = "9D 9H 9S 9C 7D".split() # Four of a Kind
    fh = "TD TC TH 7C 7D".split() # Full House
    assert poker([sf, fk, fh]) == sf
    assert poker([fk, fh]) == fk
    assert poker([fh, fh]) == fh
    assert poker([sf]) == sf
    assert poker([sf] + 99*[fh]) == sf
    assert hand_rank(sf) == (8,10)
    assert hand_rank(fk) == (7,9,7)
    assert hand_rank(fh) == (6,10,7)
print test()
```

Writing Hand Rank

```
def hand_rank(hand):
    ranks = card_ranks(hand)
    if straight(ranks) and flush(hand):            # straight flush
        return (8, max(ranks))
    elif kind(4, ranks):                           # 4 of a kind
        return (7, kind(4, ranks), kind(1, ranks))
    elif kind(3, ranks) and kind(2, ranks):        # full house
        return (6, kind(3, ranks), kind(2, ranks))
    elif flush(hand):                              # flush
        return (5, ranks)
    elif straight(ranks):                          # straight
        return (4, max(ranks))
    elif kind(3, ranks):                           # 3 of a kind
        return (3, kind(3, ranks),ranks)
    elif two_pair(ranks):                          # 2 pair
        return (2, two_pair(ranks),ranks)
    elif kind(2, ranks):                           # kind
        return (1, kind(2, ranks),ranks)
    else:                                          # high card
        return (0, ranks)
```

Testing Card Rank

```
def test():
    "Test cases for the functions in poker program"
    sf = "6C 7C 8C 9C TC".split() # Straight Flush
    fk = "9D 9H 9S 9C 7D".split() # Four of a Kind
    fh = "TD TC TH 7C 7D".split() # Full House
    assert card_ranks(sf)==[10,9,8,7,6]
    assert card_ranks(fk)==[9,9,9,9,7]
    assert card_ranks(fh)==[10,10,10,7,7]
    assert poker([sf, fk, fh]) == sf
    assert poker([fk, fh]) == fk
    assert poker([fh, fh]) == fh
    assert poker([sf]) == sf
    assert poker([sf] + 99*[fh]) == sf
    assert hand_rank(sf) == (8, 10)
    assert hand_rank(fk) == (7, 9, 7)
    assert hand_rank(fh) == (6, 10, 7)
    return 'tests pass'
```

Fixing Card Rank

```
def card_ranks(hand):
    "Return a list of the ranks, sorted with a higher first."
    ranks = ['--23456789TJQKA'.index(r) for r,s in hand]
    ranks.sort(reverse=True)
    return ranks

print card_ranks(['AC', '3D', '4S', 'KH']) #should output [14, 13, 4, 3]
```

Straight And Flush

```
def straight(ranks):
    "Return True if the ordered ranks form a 5-card straight."
    return (max(ranks)-min(ranks) == 4) and len(set(ranks))==5

def flush(hand):
    "Return True if all the cards have the same suit."
    suits = [s for r,s in hand]
    return len(set(suits))== 1
```

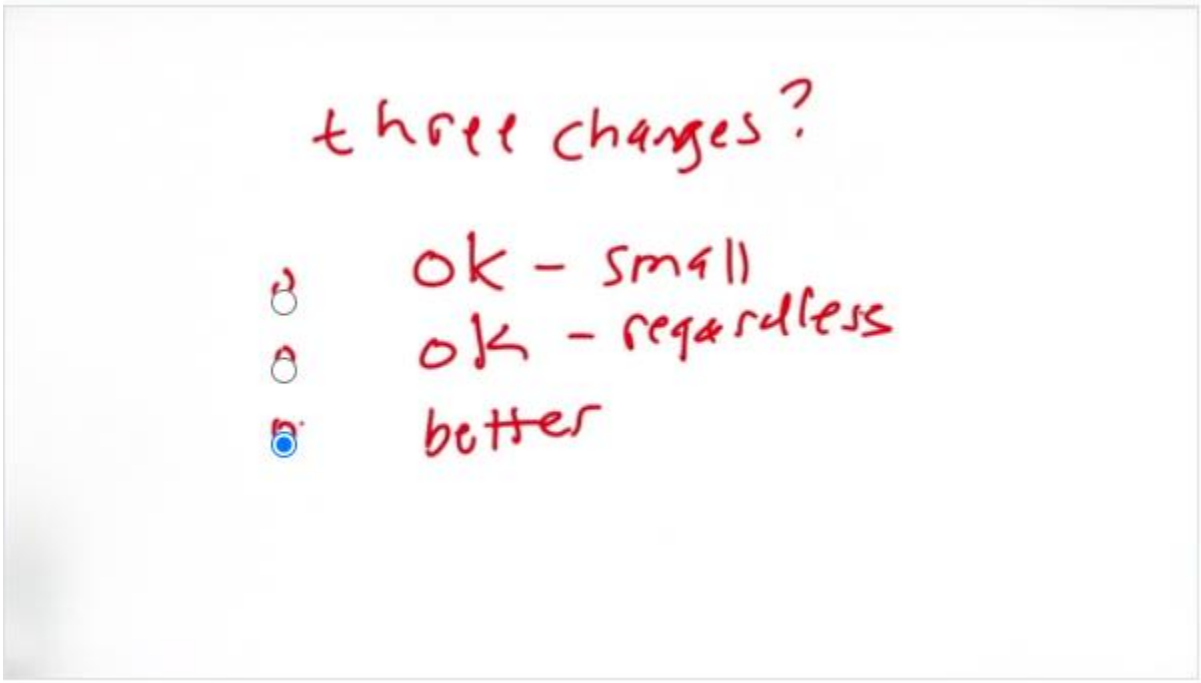
Kind function

```
6  def kind(n, ranks):
7      """Return the first rank that this hand has exactly n of.
8      Return None if there is no n-of-a-kind in the hand."""
9      for r in ranks:
10         if ranks.count(r) == n: return r
11     return None
```

Two Pair Function

```
def two_pair(ranks):
    """If there are two pair, return the two ranks as a
    tuple: (highest, lowest); otherwise return None."""
    pair=kind(2,ranks)
    lowpair=kind(2,list(reversed(ranks)))
    if pair and lowpair != pair:
        return (pair, lowpair)
    else:
        return None
```

Making Changes

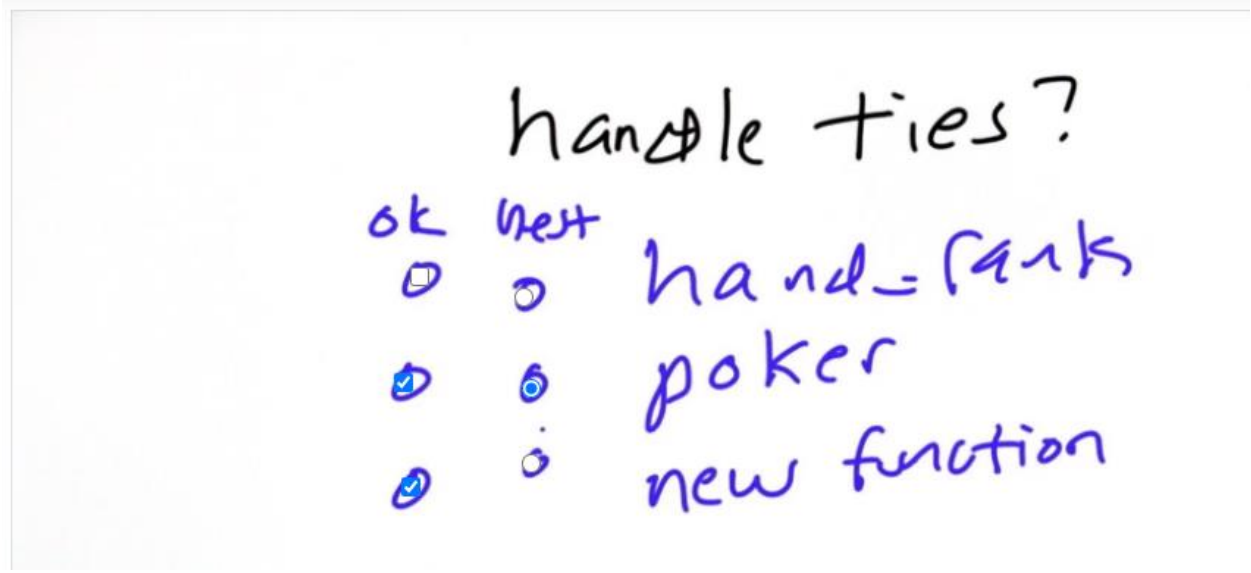


What to Change

Ace Low Straight

```
def card_ranks(hand):
    "Return a list of the ranks, sorted with higher first."
    ranks = ['--23456789TJQKA'.index(r) for r, s in hand]
    ranks.sort(reverse = True)
    return [5,4,3,2,1] if (ranks==[14,5,4,3,2]) else ranks
```

Handling Ties



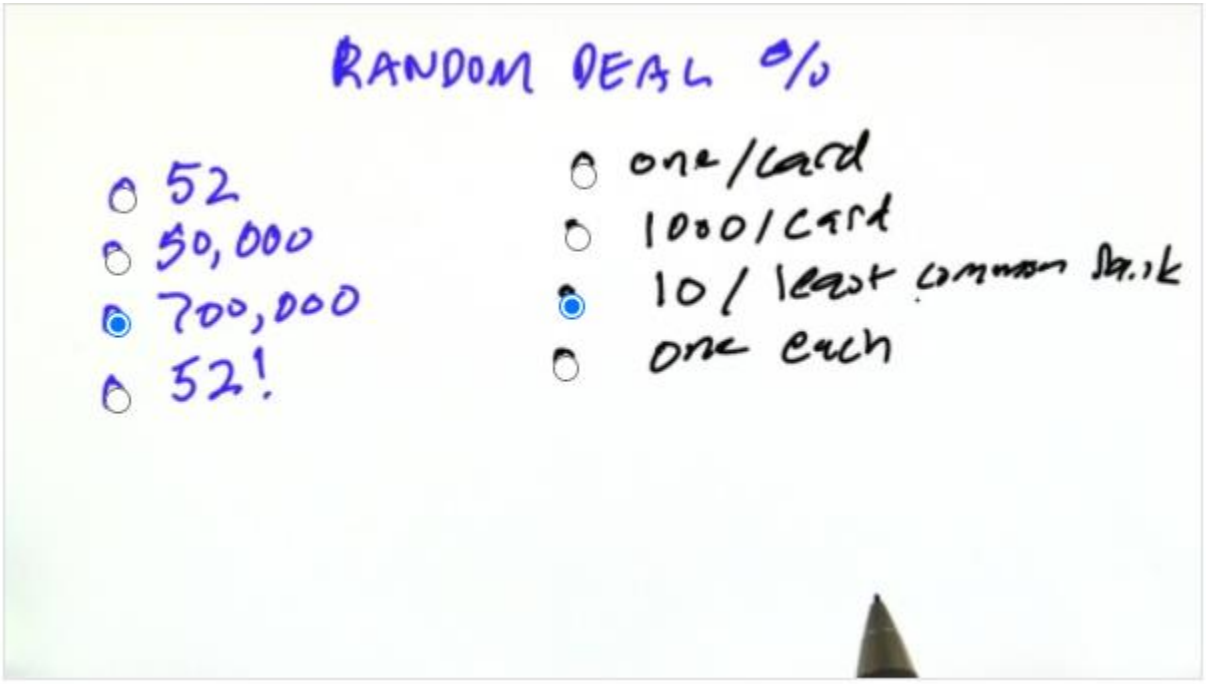
Allmax

```python
def allmax(iterable, key=None):
    "Return a list of all items equal to the max of the iterable."
    result, maxval =[], None
    key = key or (lambda x: x)
    for x in iterable:
        xval = key(x)
    if not result or xval > maxval:
        result, maxval = [x], xval
    elif xval == maxval:
        result.append(x)
    return result
```

Deal

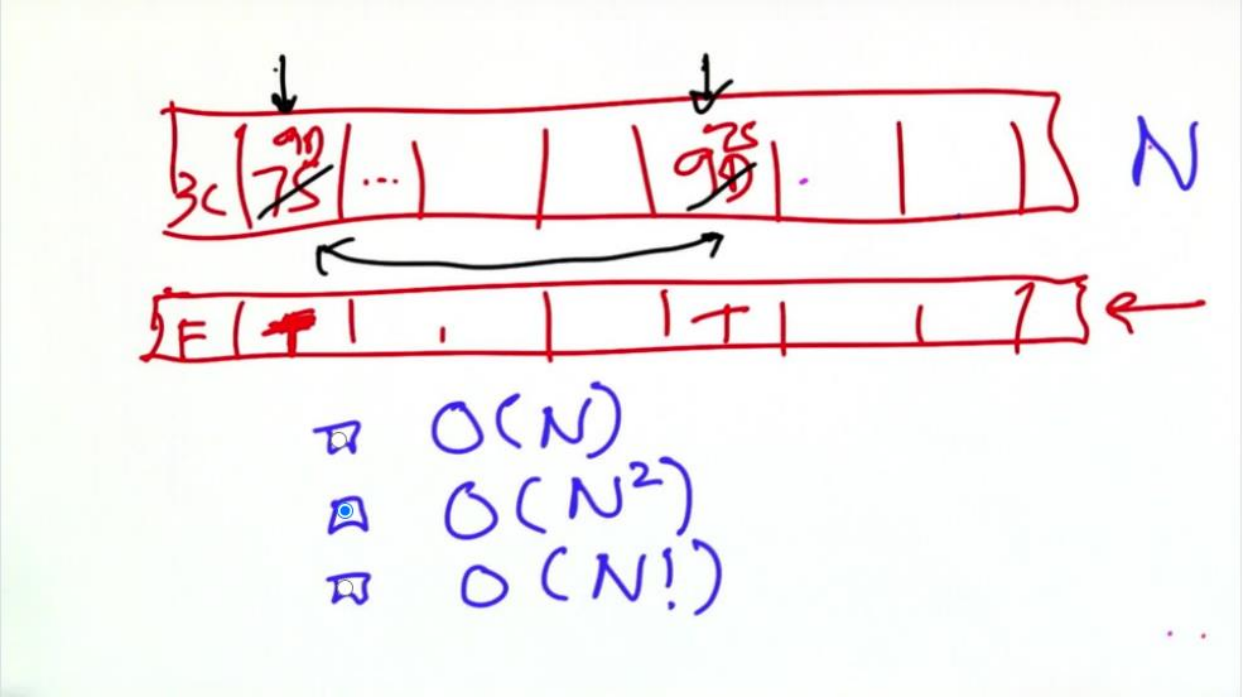
```python
def deal(numhands, n=5, deck=mydeck):
    random.shuffle(deck)
    return [deck[n*i:n*(i+1)] for i in range(numhands)]
```

Hand Frequencies

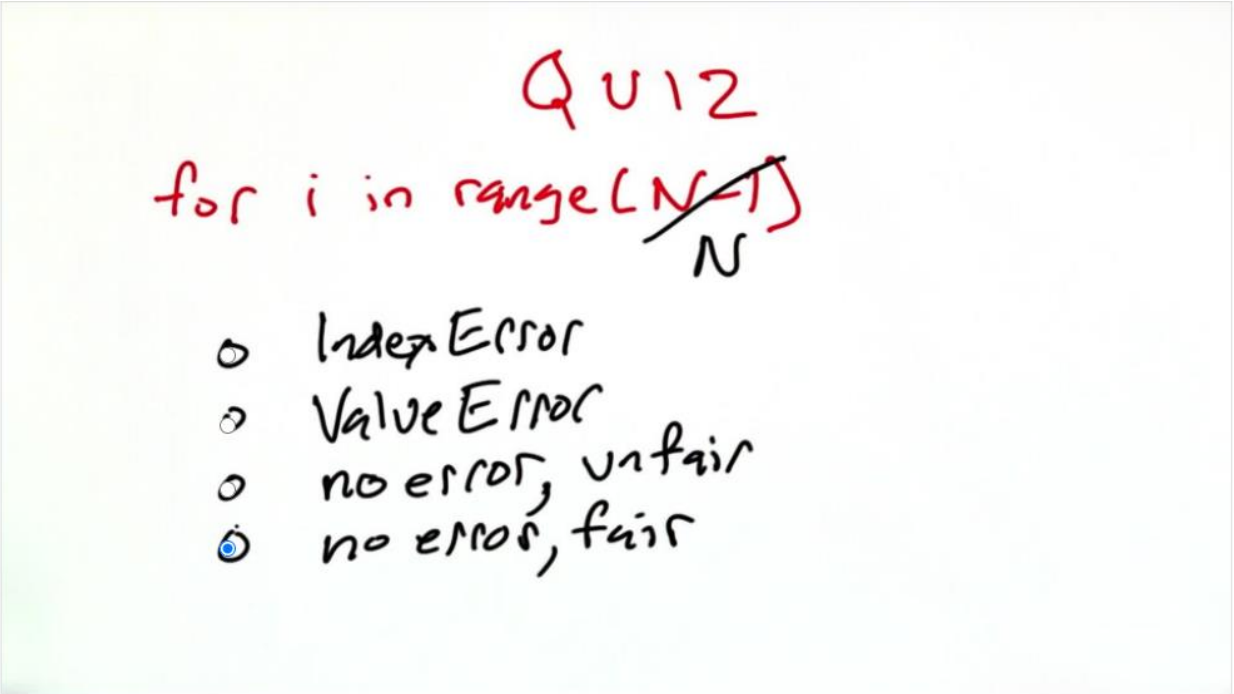


LECCIÓN 2

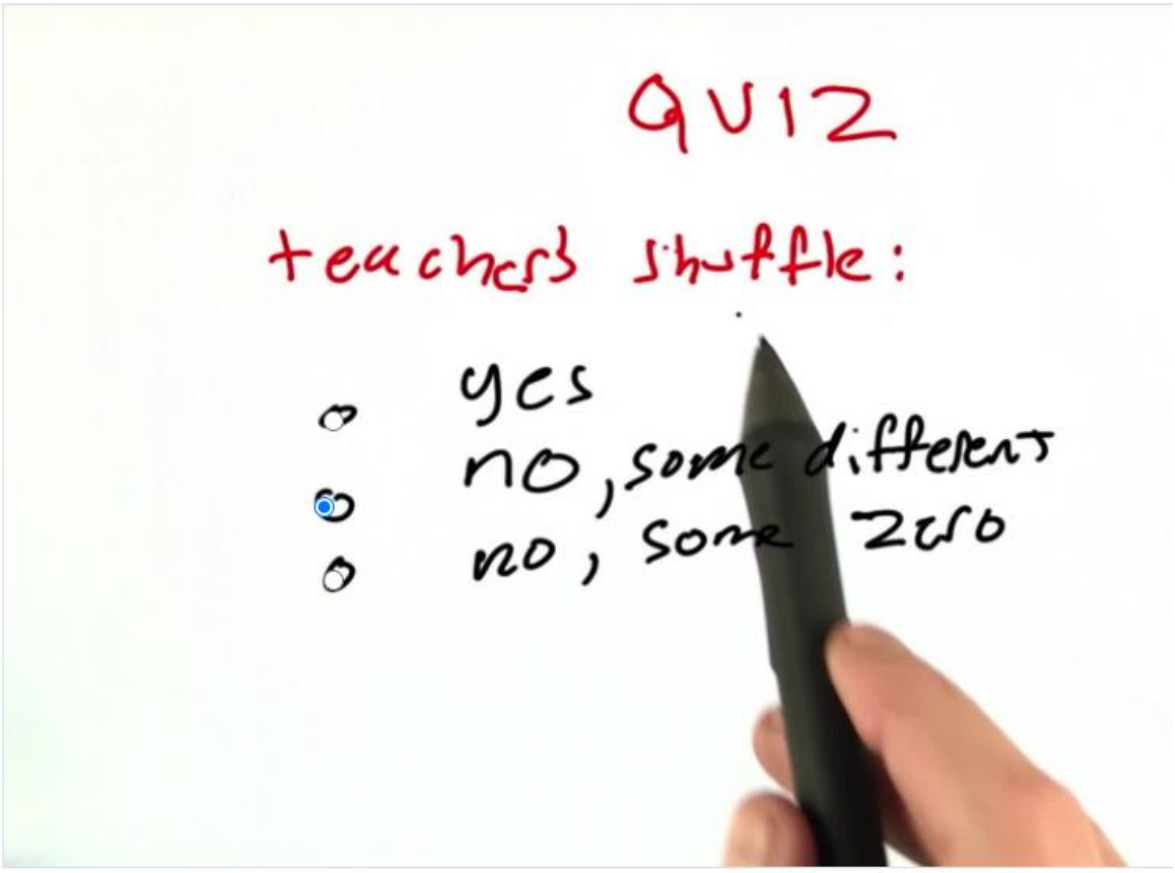
Shuffle Runtime



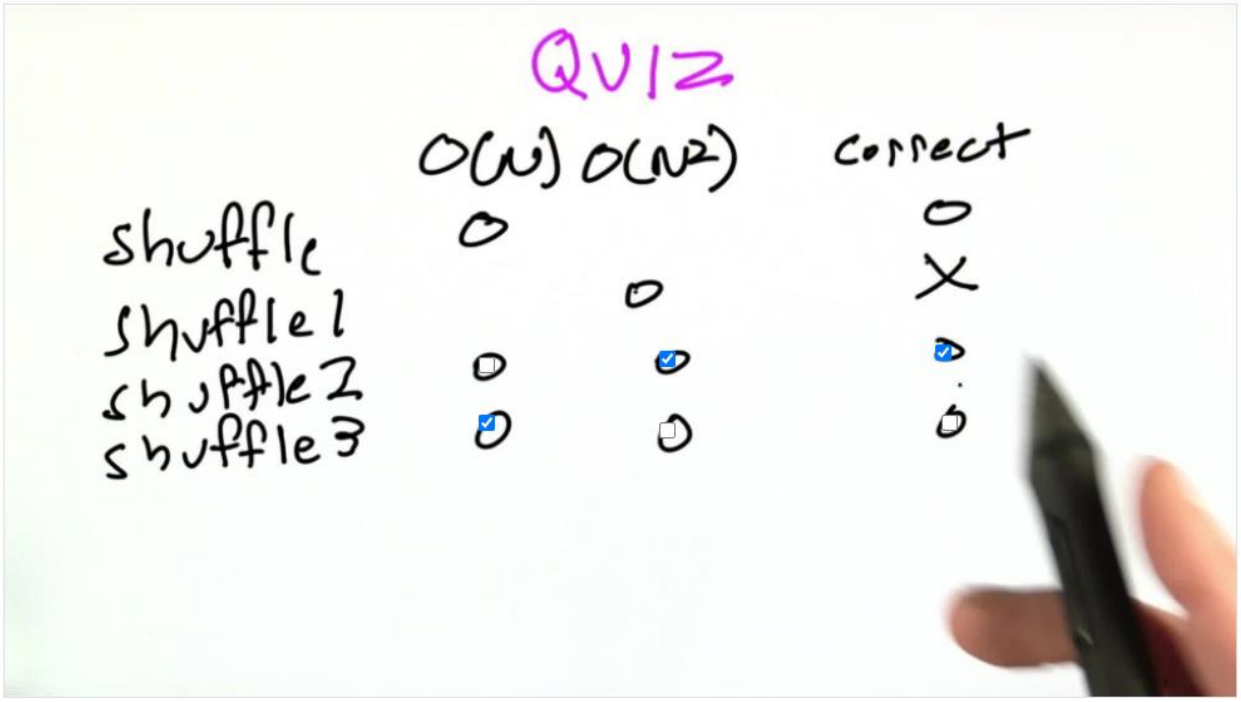
Good Shuffle



Is it Random

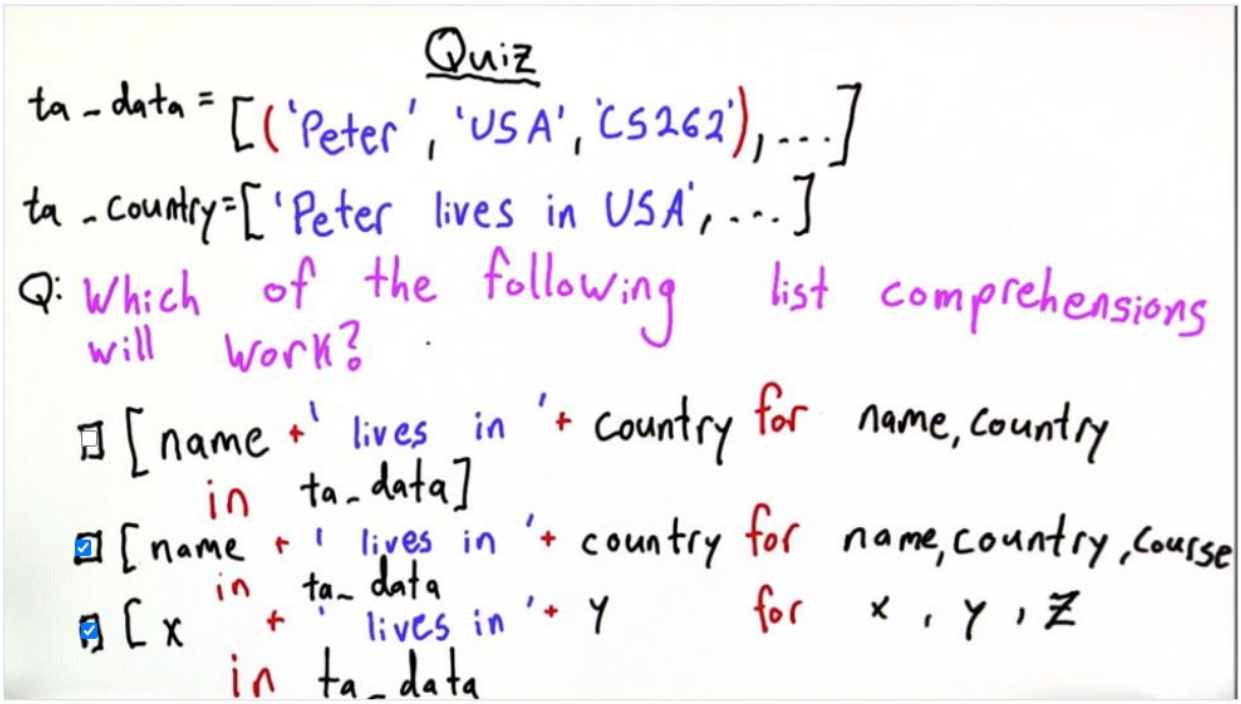


Comparing Shuffles


QUIZ
O(N) O(N²) correct
shuffle
shuffle1
shuffle2
shuffle3
X

LECCIÓN 3

List Comprehensions 2



List Comprehensions 4

```
ta_300 = [name + ' is the TA for ' + course for name, country,
          course in ta_data if course.find('CS3') != -1]
```

LECCIÓN 4

Seven Card Stud

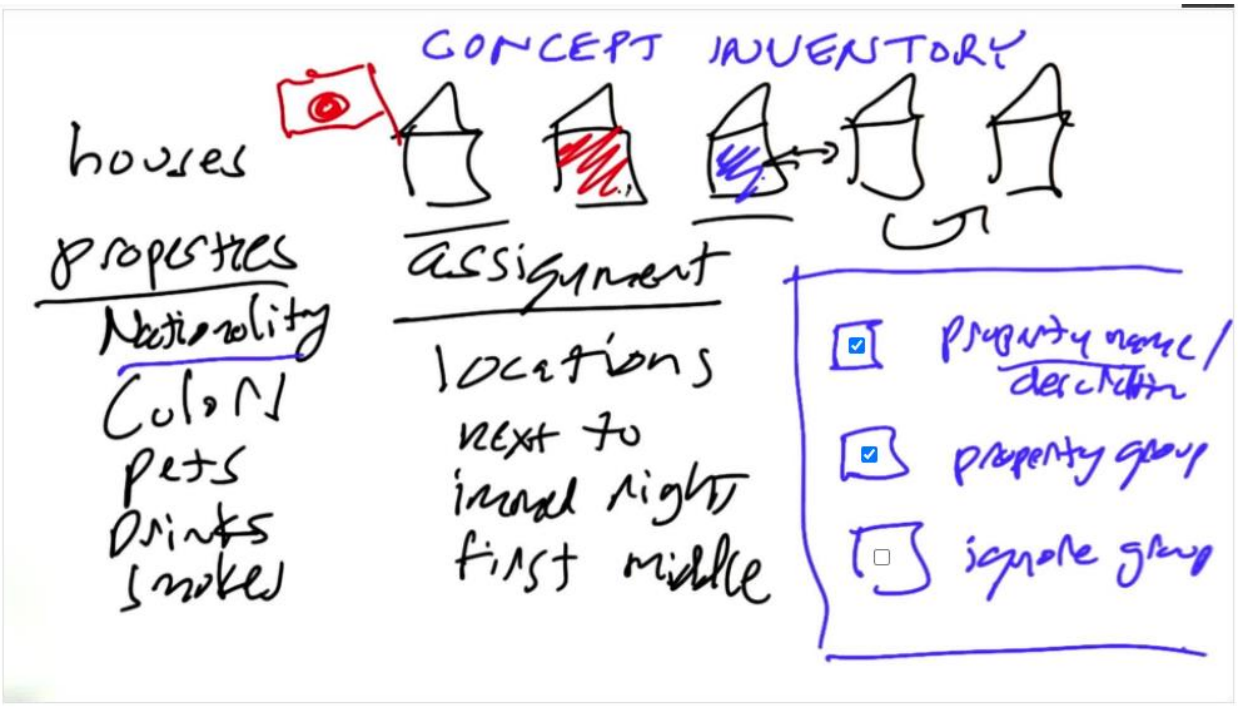
```
19  import itertools
20
21  def best_hand(hand):
22      "From a 7-card hand, return the best 5 card hand."
23      return max(itertools.combinations(hand, 5), key=hand_rank)
24
```

Jokers Wild

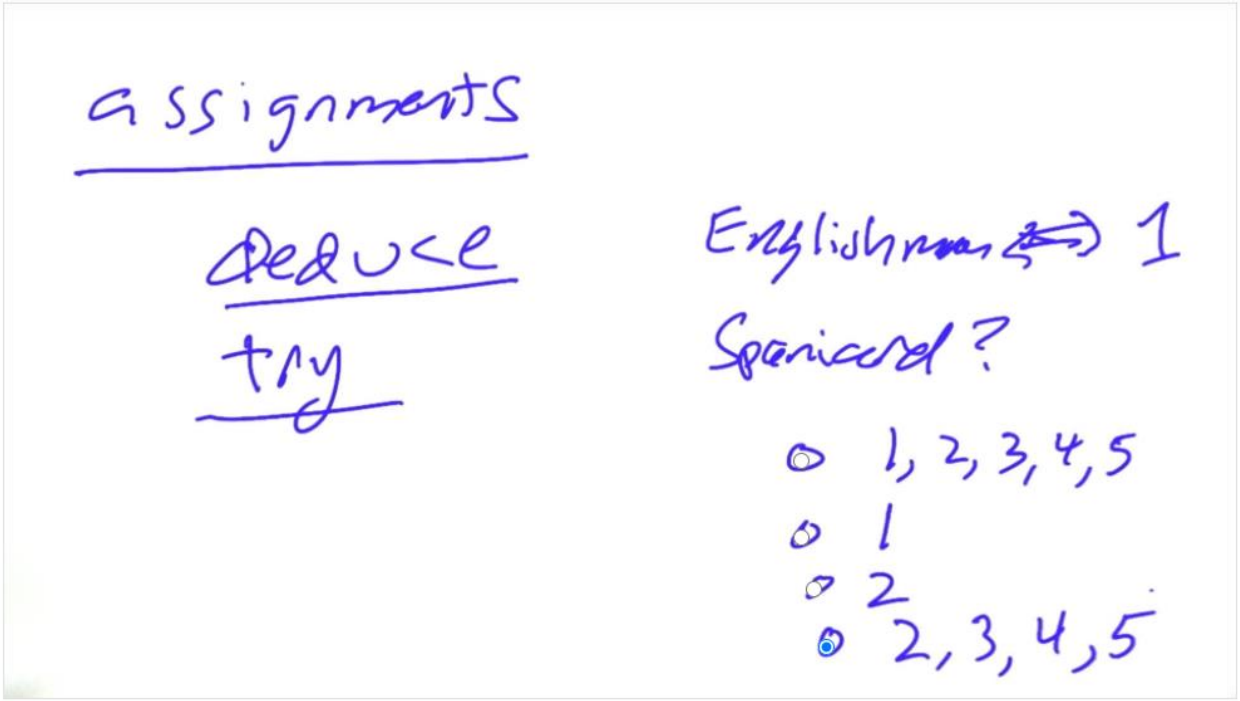
```
def best_wild_hand(hand):
    "Try all values for jokers in all 5-card selections."
    hands = set(best_hand(h)
                for h in itertools.product(*map(replacements,hand)))
    return max(hands, key=hand_rank)

def replacements(card):
    if card =='?B':return blackcards
    elif card == '?R':return redcards
    else: return [card]
```

LECCIÓN 6

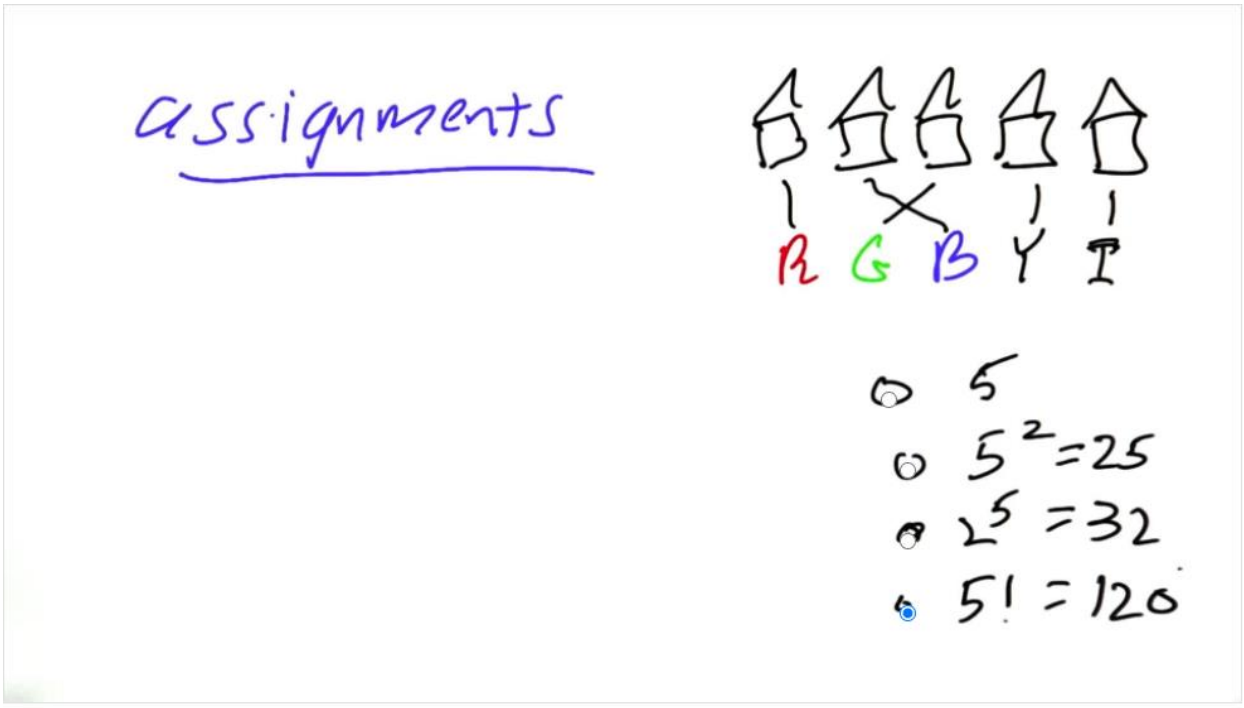
Zebra Puzzle



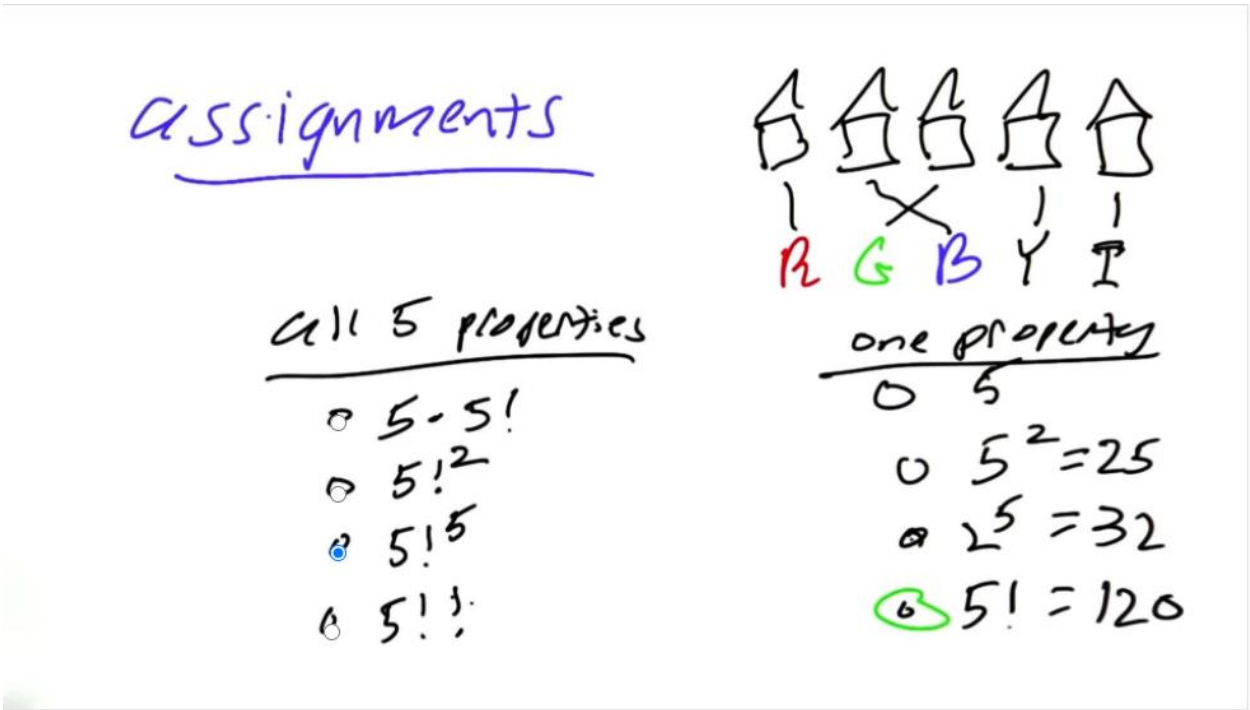
Where The Spaniard



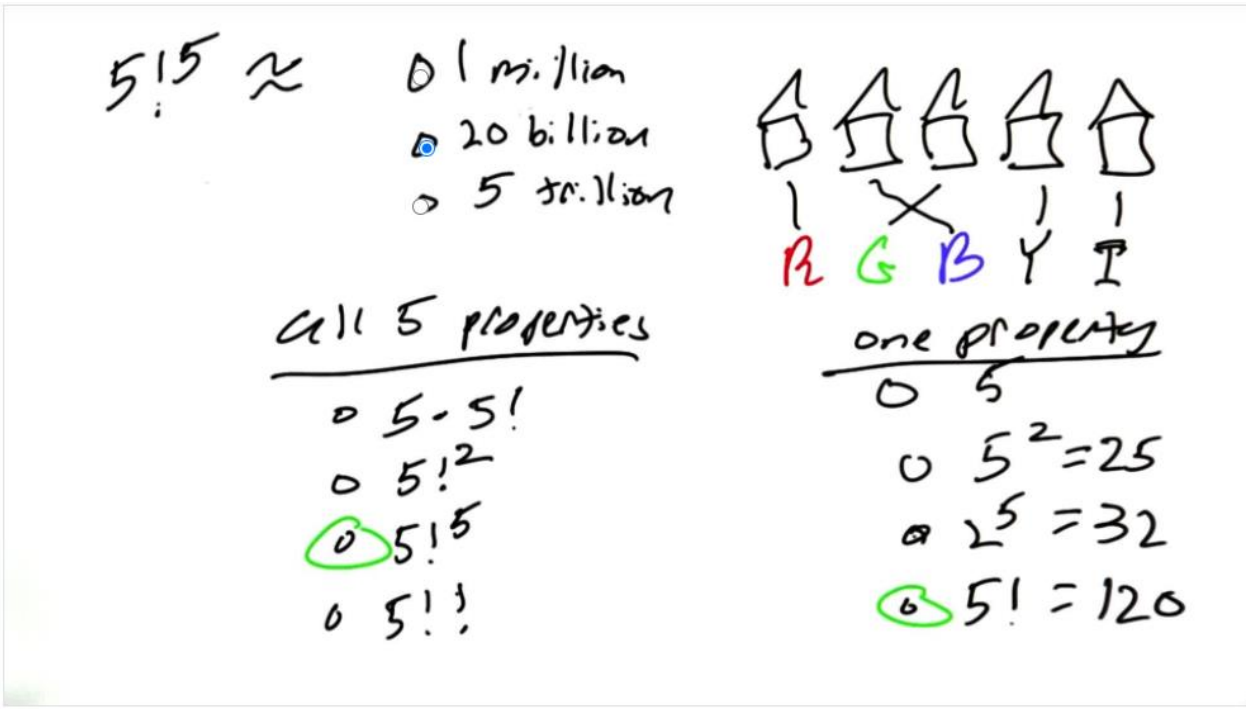
Counting Assignments



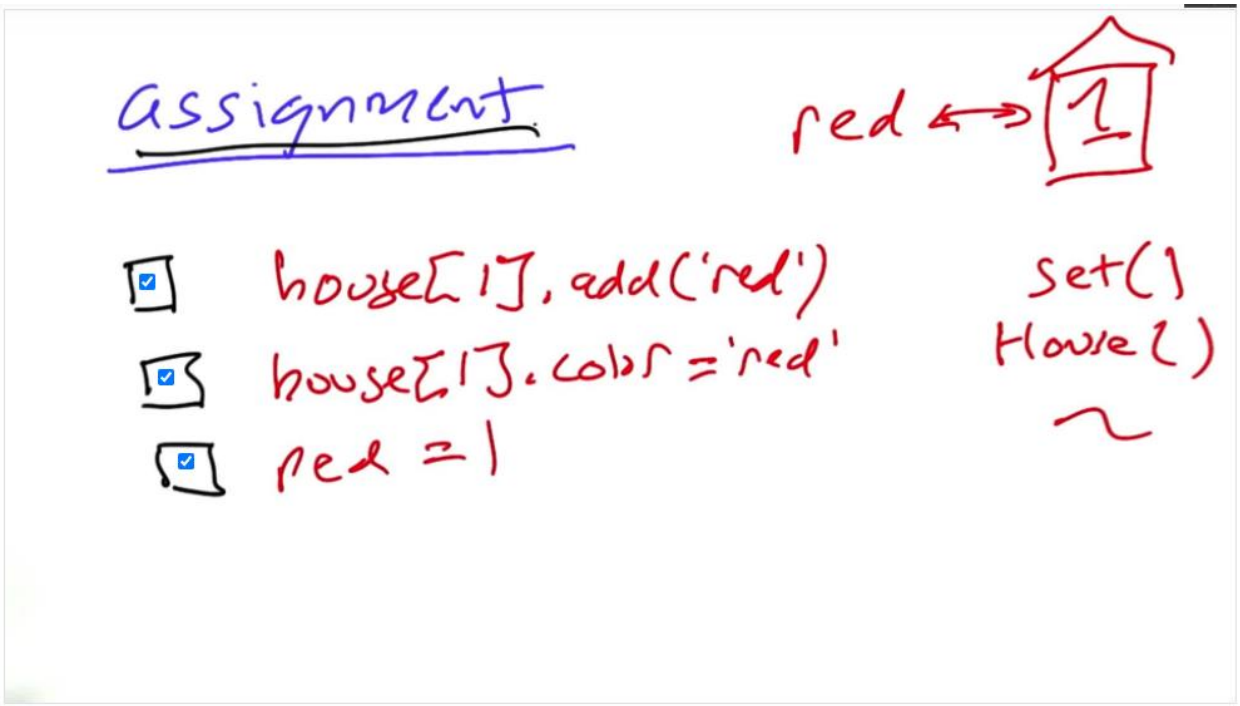
Multiple Properties



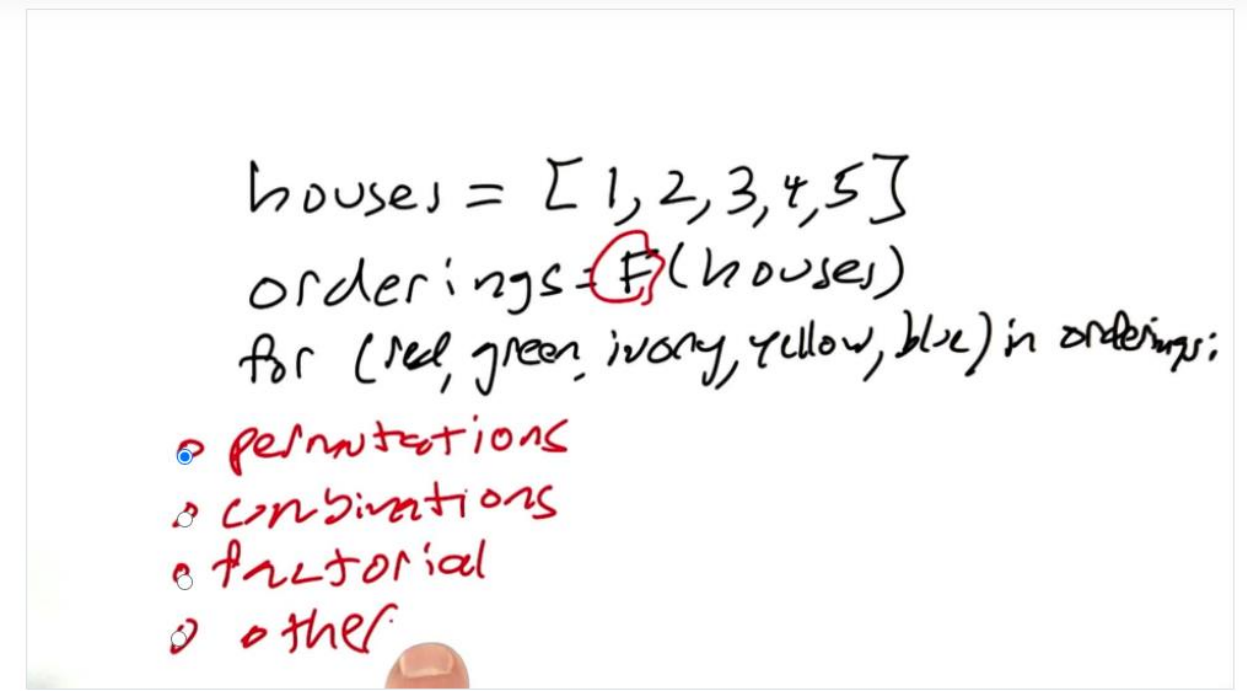
Back of the Envelope



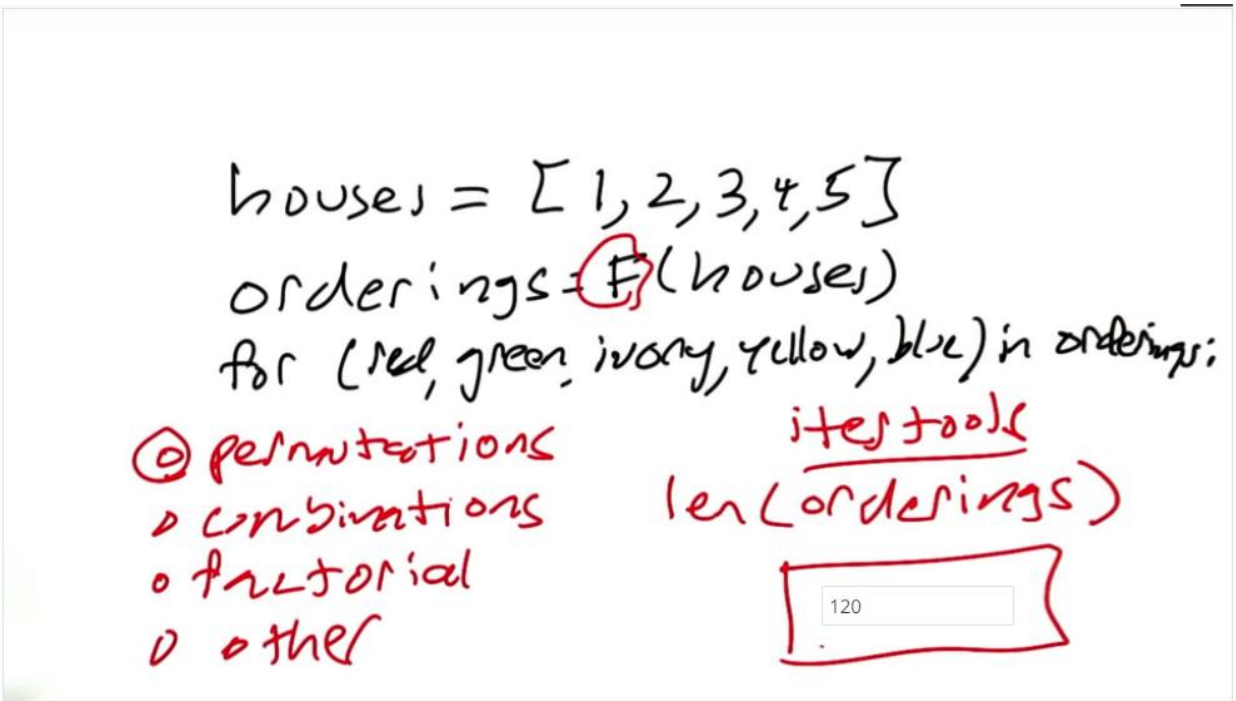
Leaving Happy Valley



Ordering Houses



Length Of Orderings



Estimating Runtime

```
1
2
3
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           # constraints go here
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

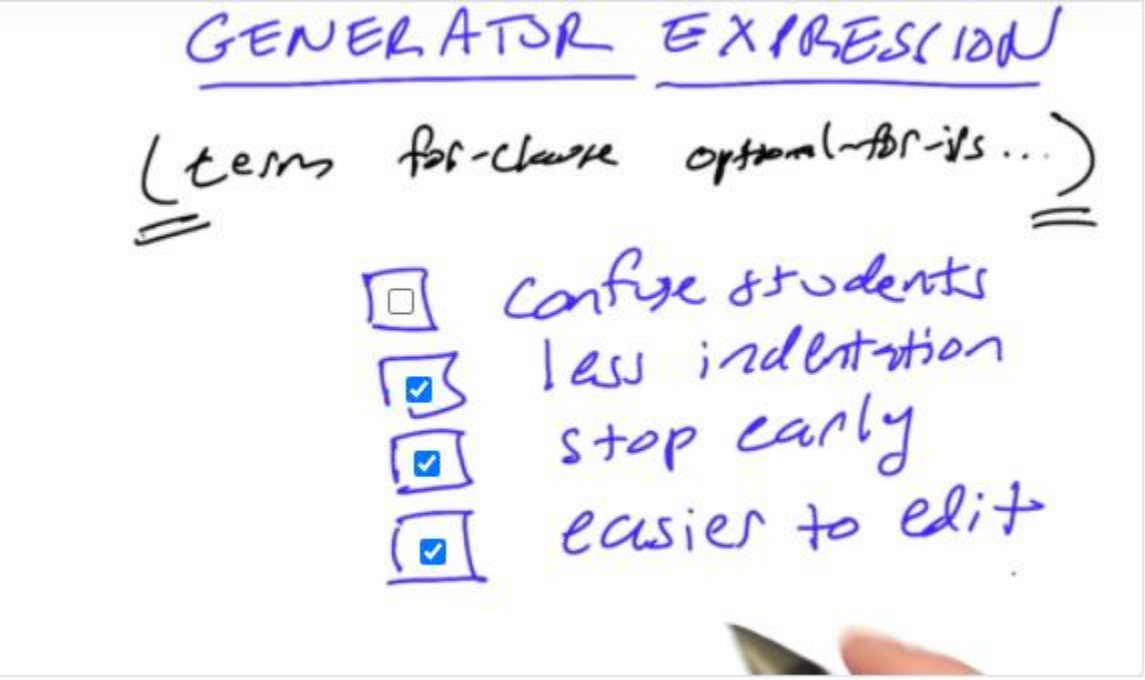
Red Englishman

```
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           if ( Englishman == red ): #2
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

Neighbors

```
 7
 8  import itertools
 9
10  houses = [1, 2, 3, 4, 5]
11  orderings = list(itertools.permutations(houses))
12
13  def imright(h1, h2):
14      "House h1 is immediately right of h2 if h1-h2 == 1."
15      return h1-h2 == 1
16
17  def nextto(h1, h2):
18      "Two houses are next to each other if they differ by 1."
19      return abs(h1-h2) == 1
20
```

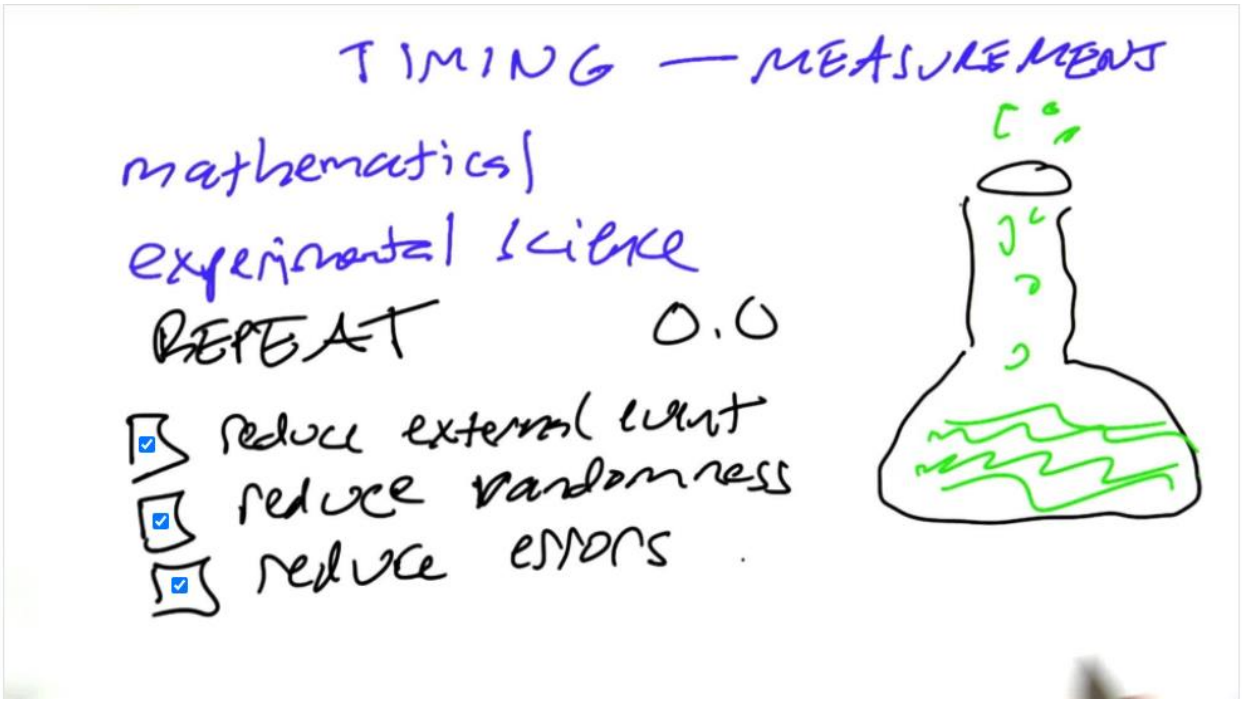
Generator Expressions



Eliminating Redundancy

```
def zebra_puzzle():
    "Return a tuple (WATER, ZEBRA) indicating their house numbers."
    houses = first, _, middle, _, _ = [1, 2, 3, 4, 5]
    orderings = list(itertools.permutations(houses))  #1
    return next((WATER, ZEBRA)
            for (red, green, ivory, yellow, blue) in orderings
            for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings
            if Englishman is red             #2
            for (dog, snails, fox, horse, ZEBRA) in orderings
            for (coffee, tea, milk, oj, WATER) in orderings
            for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings
            if Spaniard is dog               #3
```

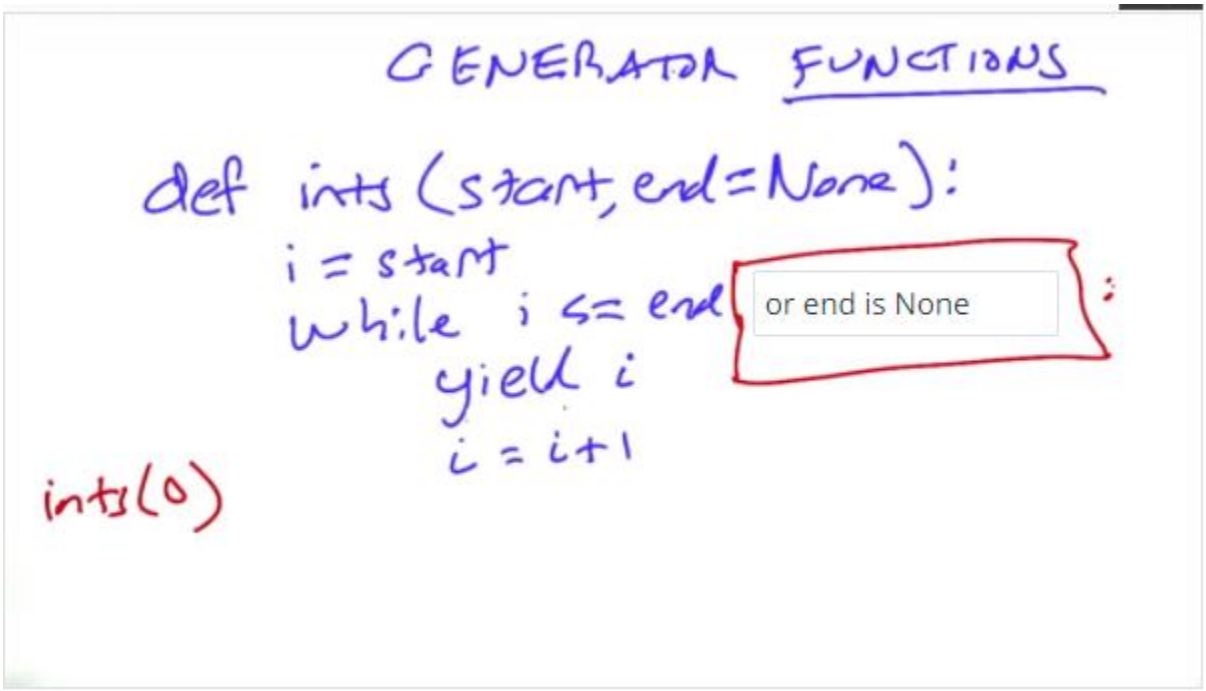
Good Science



Timed Calls

```
def timedcalls(n, fn, *args):
    """Call fn(*args) repeatedly: n times if n is an int, or up to
    n seconds if n is a float; return the min, avg, and max time"""
    if isinstance(n, int):
        times = [timedcall(fn, *args)[0] for _ in range(n)]
    else:
        times = []
        while sum(times) < n:
            times.append(timedcall(fn, *args)[0])
    return min(times), average(times), max(times)
```

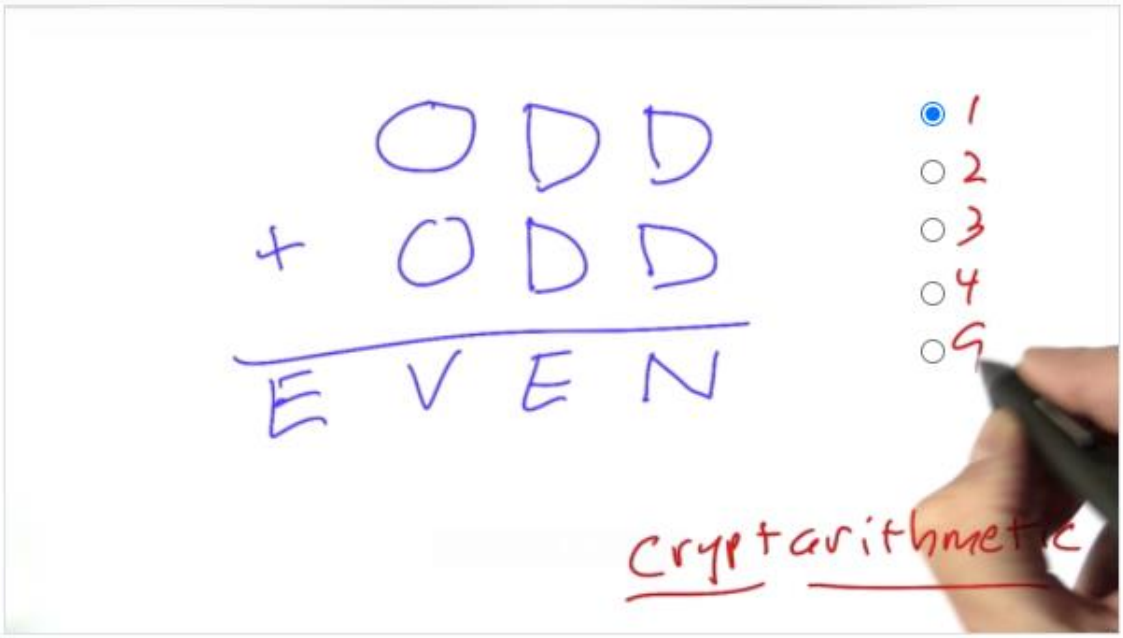
Yeilding Results



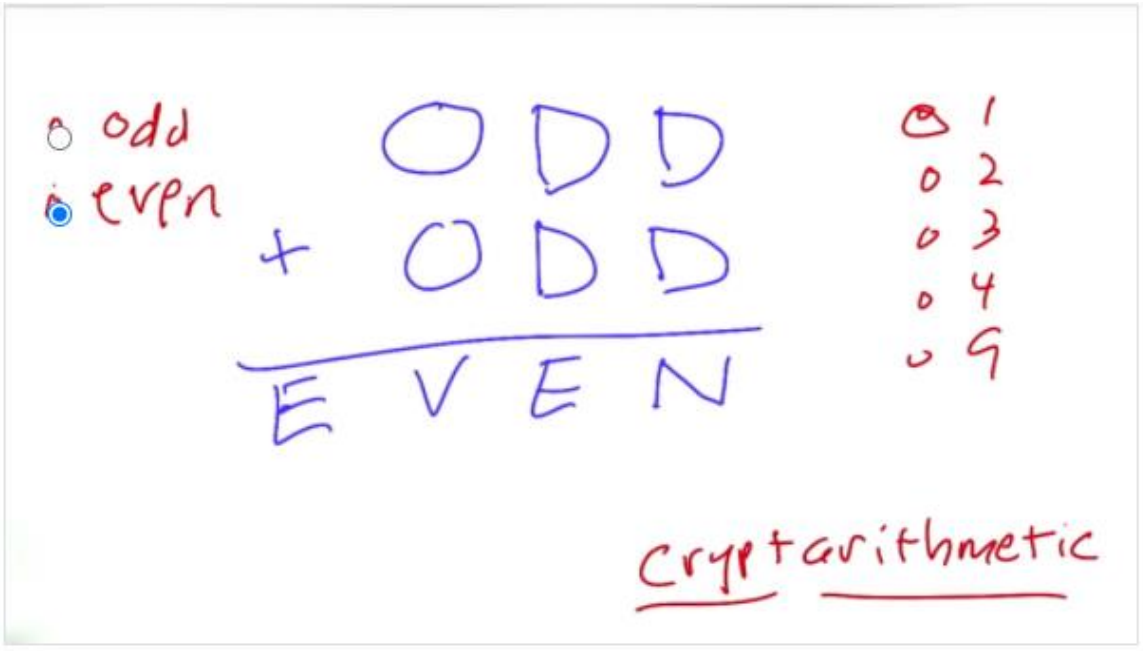
All Ints

```
14 def all_ints():
15     "Generate integers in the order 0, +1, -1, +2, -2, +3, -3, ..."
16     yield 0
17     for i in ints(1):
18         yield +i
19         yield -i
```

Cryptarithmetic



Odd or Even



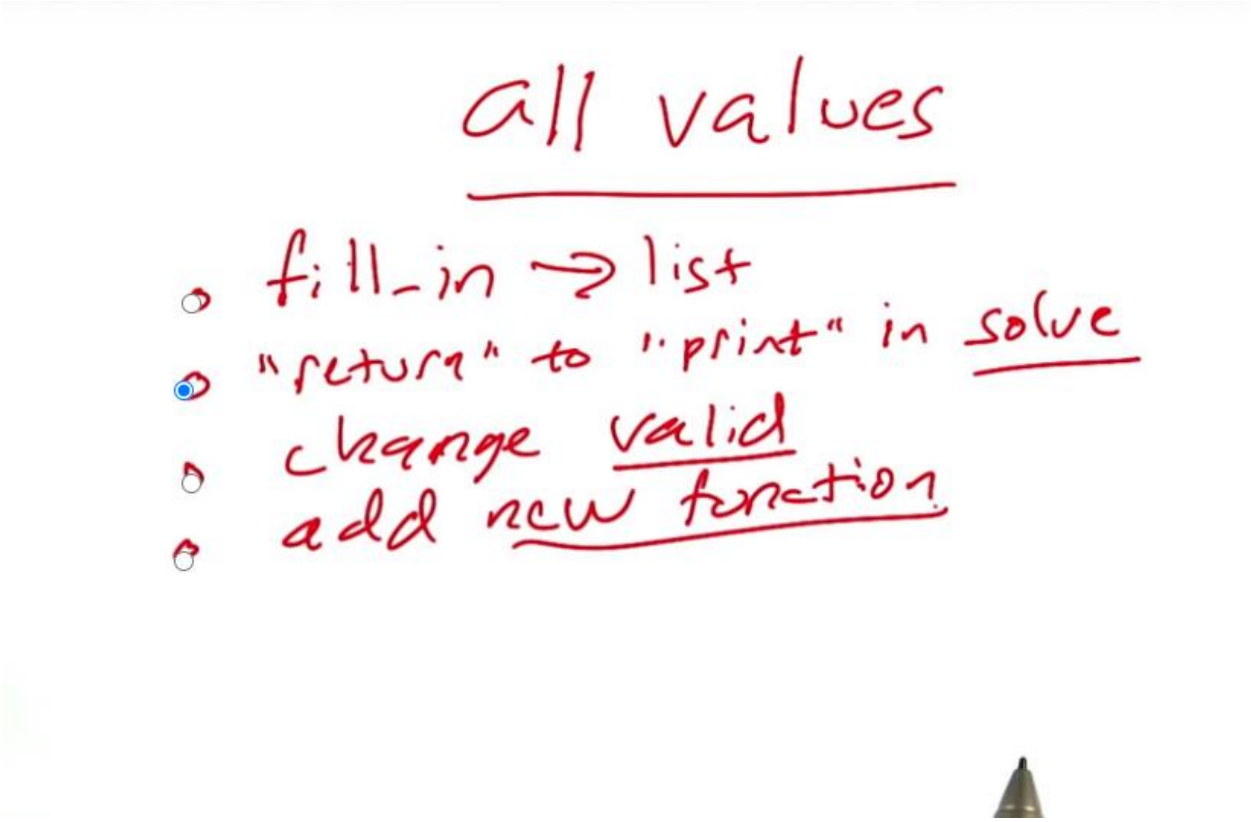
Solving Cryptarithmetic

```
def solve(formula):
    """Given a formula like 'ODD + ODD == EVEN', fill in digits to
    Input formula is a string; output is a digit-filled-in string
    for f in fill_in(formula):
        if valid(f):
            return f
```

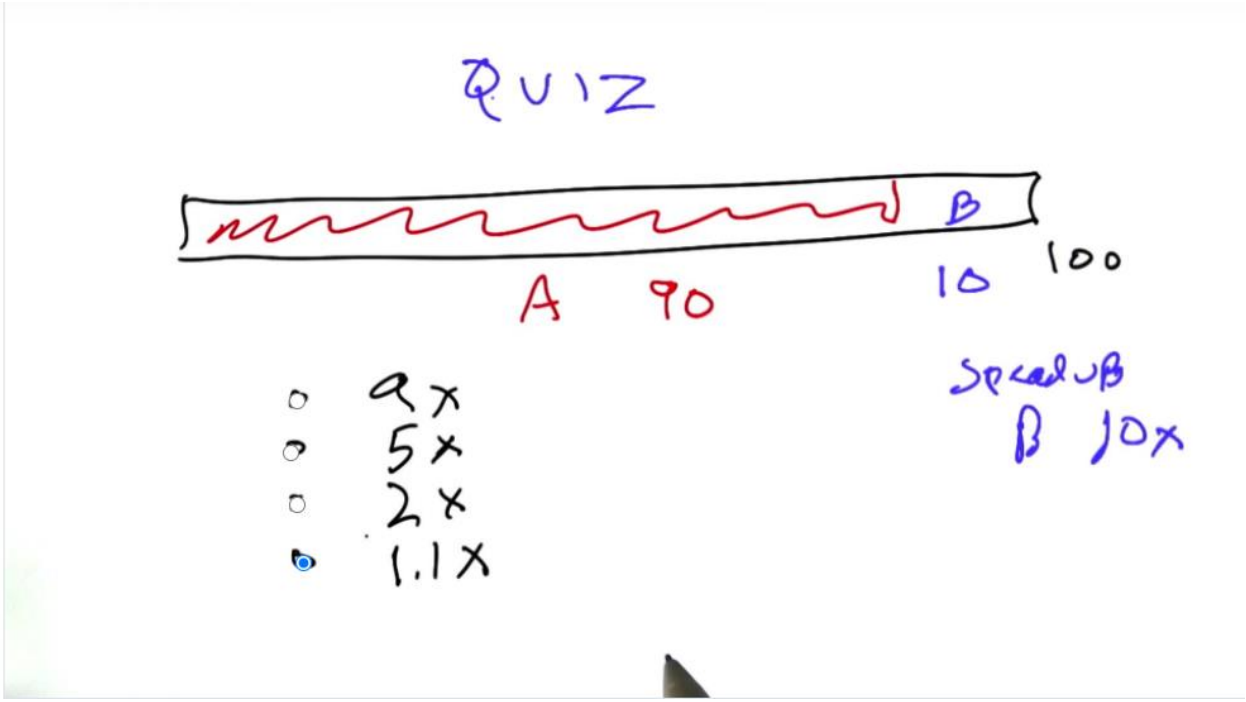
Filling In Fill In

```
def fill_in(formula):
    "Generate all possible fillings-in of letters in formula with digits."
    letters = ''.join(set(re.findall('[A-Z]', formula))) #should be a string
    for digits in itertools.permutations('1234567890', len(letters)):
        table = string.maketrans(letters, ''.join(digits))
        yield formula.translate(table)
```

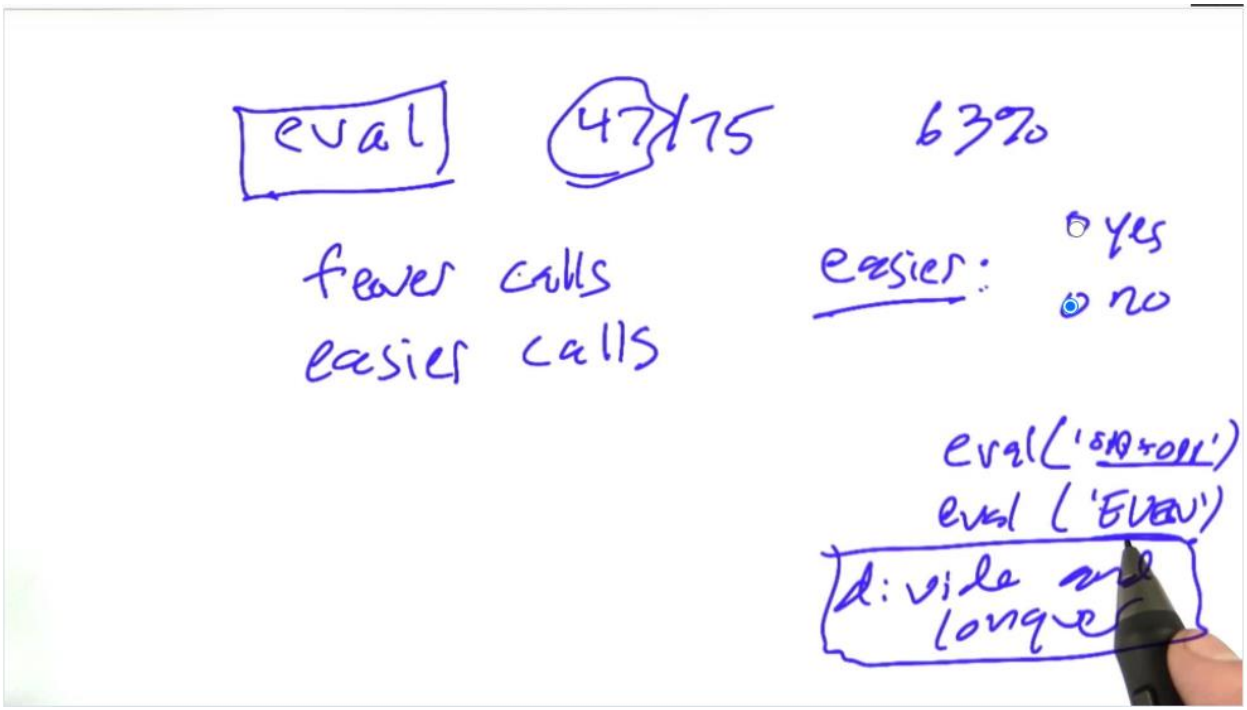
Find All Values



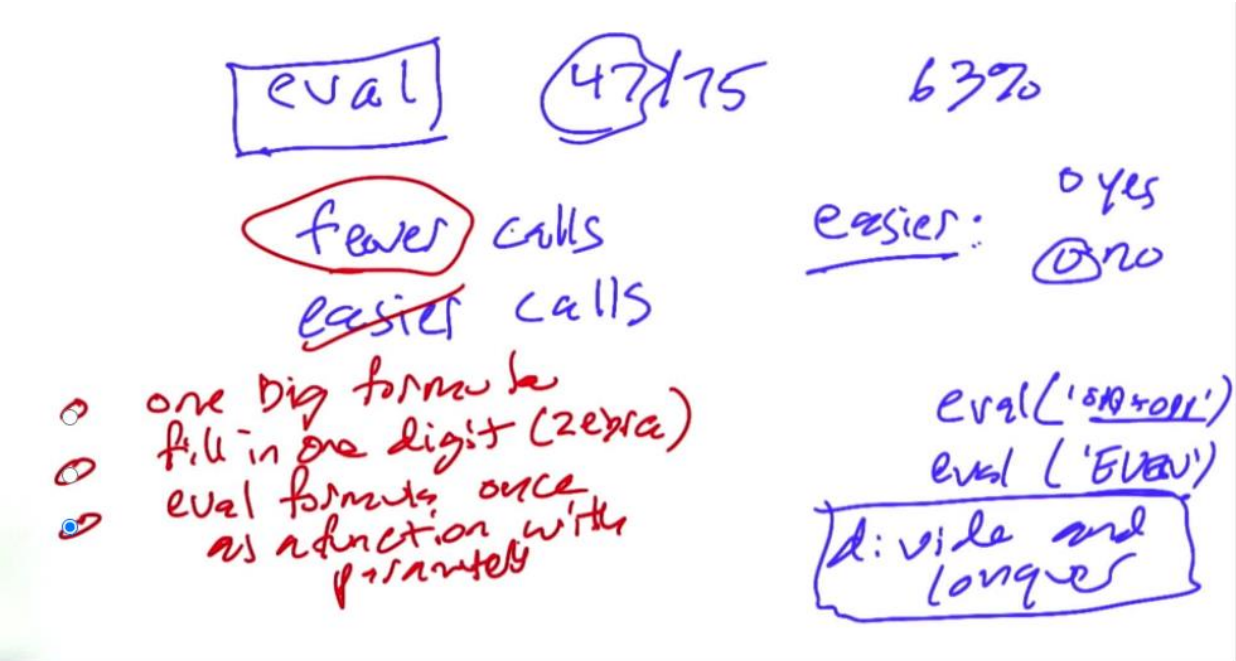
Increasing Speed



Rethinking Eval



Making Fewer Calls



Compile World

```
def compile_word(word):
    """Compile a word of uppercase letters as numeric digits.
    E.g., compile_word('YOU') => '(1*U+10*O+100*Y)'
    Non-uppercase words unchanged: compile_word('+') => '+'"""
    if word.isupper():
        terms = [('%s*%s' % (10**i, d))
                 for (i, d) in enumerate(word[::-1])]
        return '('+ '+' .join(terms) + ')'
    else:
        return word
```

LECCIÓN 7

No Leading Zeros

```
def compile_formula(formula, verbose=False):
    """Compile formula into a function. Also return letters found
    in same order as parms of function. The first digit of a mult
    number can't be 0. So if YOU is a word in the formula, and th
    is called with Y eqal to 0, the function should return False

    # modify the code in this function.

    letters = ''.join(set(re.findall('[A-Z]', formula)))
    firstletters = set(re.findall(r'\b([A-Z])[A-Z]', formula))
    parms = ', '.join(letters)
    tokens = map(compile_word, re.split('([A-Z]+)', formula))
    body = ''.join(tokens)
    if firstletters:
        test = ' and '.join(L+'!=0' for L in firstletters)
        body = '%s and (%s)' % (test, body)
    f = 'lambda %s: %s' % (parms, body)
    if verbose: print f
    return eval(f), letters
```

Floor Puzzle

```
import itertools

def floor_puzzle():
    floors = bottom,_,_,_,top=[1,2,3,4,5]
    orderings=list(itertools.permutations(floors))
    for (Hopper,Kay,Liskov,Perlis,Ritchie) in orderings:
        if(Hopper is not top
            and Kay is not bottom
            and Liskov is not top
            and Liskov is not bottom
            and Perlis > Kay
            and abs(Ritchie - Liskov)>1
            and abs(Liskov - Kay)>1):
            return [Hopper, Kay, Liskov, Perlis, Ritchie]

print floor_puzzle()
```

Subpalindrome

```
def longest_subpalindrome_slice(text):
    "Return (i, j) such that text[i:j] is the longest palindrome ir
    if text == '': return (0,0)
    def length(slice): a,b = slice; return b-a
    candidates = [grow(text, start, end)
                  for start in range(len(text))
                  for end in (start, start+1)]
    return max(candidates, key=length)

def grow(text,start,end):
    while(start>0 and end<len(text)
          and text[start-1].upper()==text[end].upper()):
        start -= 1; end +=1
    return (start, end)
```

LECCIÓN 9

Regular Expressions Review - Quiz 1

```
def search(pattern, text):
    """Return true if pattern appears anywhere in text
       Please fill in the match(          , text) below.
       For example, match(your_code_here, text)"""
    if pattern.startswith('^'):
        return match(pattern[1:], text) # fill this line
    else:
        return match('.*' + pattern, text) # fill this line

```

Regular Expressions Review - Quiz 2

```
def match(pattern, text):
    """
    Return True if pattern appears at the start of text

    For this quiz, please fill in the return values for:
        1) if pattern == '':
        2) elif pattern == '$':
    """

    if pattern == '':
        return True
    elif pattern == '$':
        return (text == '')
    # you can ignore the following elif and else conditions
    # We'll implement them in the next quiz
    elif len(pattern) > 1 and pattern[1] in '*?':
        return True
    else:
        return True
```

Regular Expressions Review - Quiz 3

```
    if pattern == '':
        return True
    elif pattern == '$':
        return (text == '')
    elif len(pattern) > 1 and pattern[1] in '*?':
        p, op, pat = pattern[0], pattern[1], pattern[2:]
        if op == '*':
            return match_star(p, pat, text)
        elif op == '?':
            if match1(p, text) and match(pat, text[1:]):
                return True
            else:
                return match(pat, text)
    else:
        return (match1(pattern[0], text) and
                match(pattern[1:], text[1:])) # fill in this line
```

Matchset

```
def matchset(pattern, text):
    "Match pattern at start of text; return a set of remainders of
    op, x, y = components(pattern)
    if 'lit' == op:
        return set([text[len(x):]]) if text.startswith(x) else nul
    elif 'seq' == op:
        return set(t2 for t1 in matchset(x, text) for t2 in matchs
    elif 'alt' == op:
        return matchset(x, text) | matchset(y, text)
    elif 'dot' == op:
        return set([text[1:]]) if text else null
    elif 'oneof' == op:
        return set([text[1:]]) if text.startswith(x) else null
    elif 'eol' == op:
        return set(['']) if text == '' else null
    elif 'star' == op:
        return (set([text]) |
                set(t2 for t1 in matchset(x, text)
                    for t2 in matchset(pattern, t1) if t1 != text)
    else:
```

Filling Out The Api

```
def lit(string):  return ('lit', string)
def seq(x, y):    return ('seq', x, y)
def alt(x, y):    return ('alt', x, y)
def star(x):      return ('star', x)
def plus(x):      return seq(x, star(x))
def opt(x):       return alt(lit(''), x) #opt(x) me
def oneof(chars): return ('oneof', tuple(chars))
dot = ('dot',)
eol = ('eol',)
```

Search and Match

```
def search(pattern, text):
    "Match pattern anywhere in text; return longest earliest mat
    for i in range(len(text)):
        m = match(pattern, text[i:])
        if m: ''
            return m

def match(pattern, text):
    "Match pattern against start of text; return longest match f
    remainders = matchset(pattern, text)
    if remainders:
        shortest = min(remainders, key=len)
        return text[:len(text)-len(shortest)]
```

Alt

```
def lit(s): return lambda text: set([text[len(s):]]) if text.start

def seq(x, y): return lambda text: set().union(*map(y, x(text)))

def alt(x, y): return lambda text: x(text) | y(ext)
```

Simple Compilers

```
def match(pattern, text):
    "Match pattern against start of text; return
    remainders = pattern(text)
    if remainders:
        shortest = min(remainders, key=len)
        return text[:len(text)-len(shortest)]
```

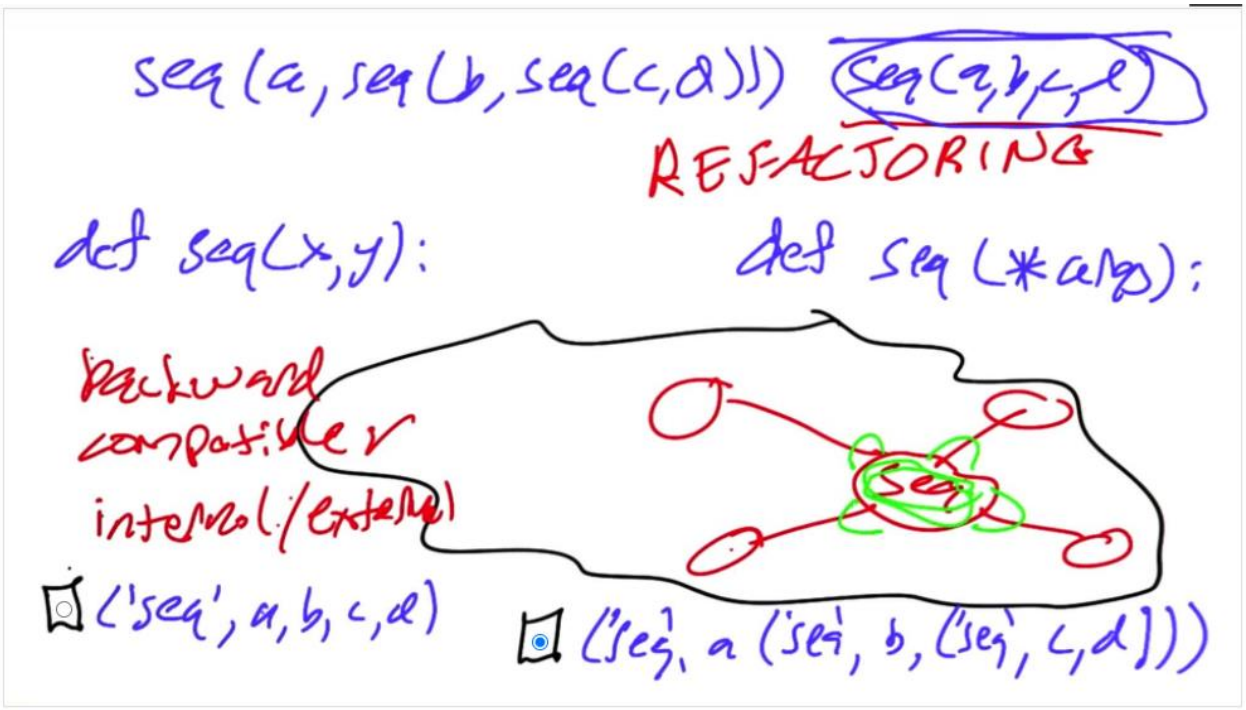
Oneof And Alt

```
def lit(s):         return lambda Ns: set([s]) if len(s) in Ns else null
def alt(x, y):      return lambda Ns: x(Ns) | y(Ns)
def star(x):        return lambda Ns: opt(plus(x))(Ns)
def plus(x):        return lambda Ns: genseq(x, star(x), Ns, startx=1) #Tricky
def oneof(chars):   return lambda Ns: set(chars) if 1 in Ns else null
def seq(x, y):      return lambda Ns: genseq(x, y, Ns)
def opt(x):         return alt(epsilon, x)
dot = oneof('?')    # You could expand the alphabet to more chars.
epsilon = lit('')   # The pattern that matches the empty string.

null = frozenset([])
```

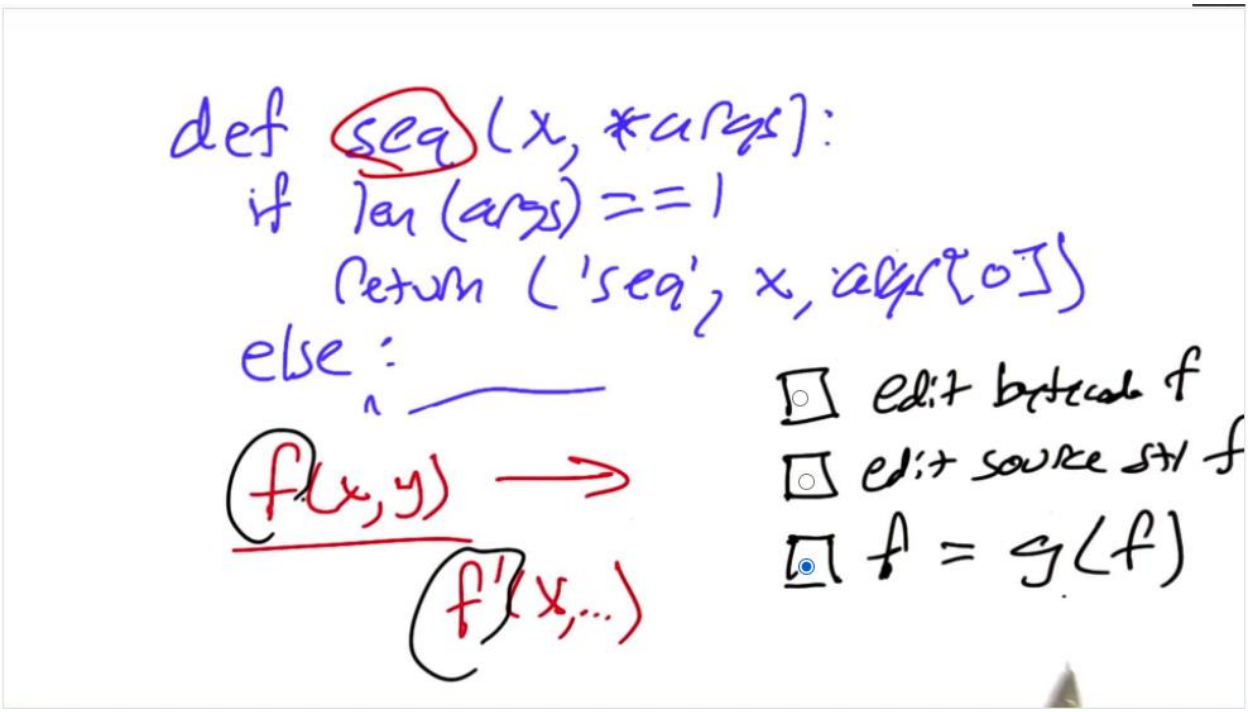
Genseq

```
3
4  def genseq(x, y, Ns):
5      Nss = range(max(Ns)+1)
6      return set(m1 + m2
7                 for m1 in x(Nss) for m2 in y(Nss)
8                 if len(m1 + m2) in Ns)
9
10
11         [○] correct for all inputs
12         [○] incorrect for some
13         [◉] correct when returns; doesn't always return
```

Changing Seq



Function Mapping



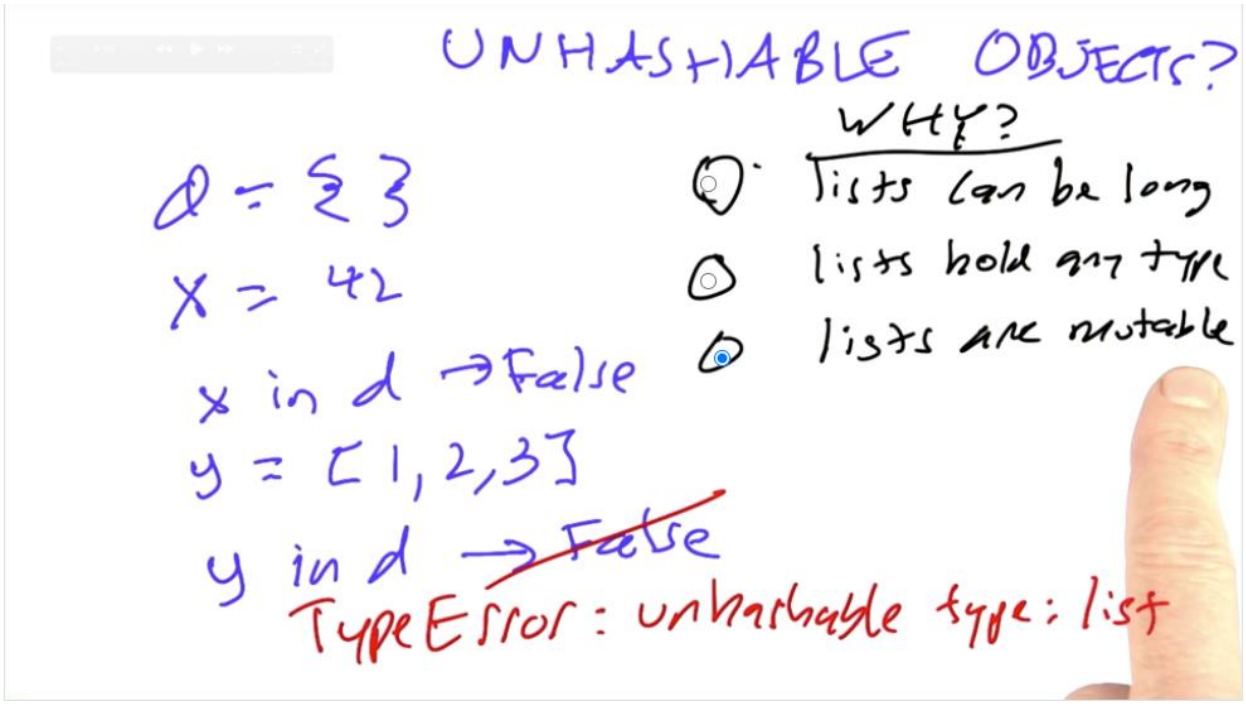
N Ary Function

```
def n_ary(f):
    """Given binary function f(x, y), return an n_ary function such
    that f(x, y, z) = f(x, f(y,z)), etc. Also allow f(x) = x."""
    def n_ary_f(x, *args):
        return x if not args else f(x, n_ary_f(*args))
    return n_ary_f
```

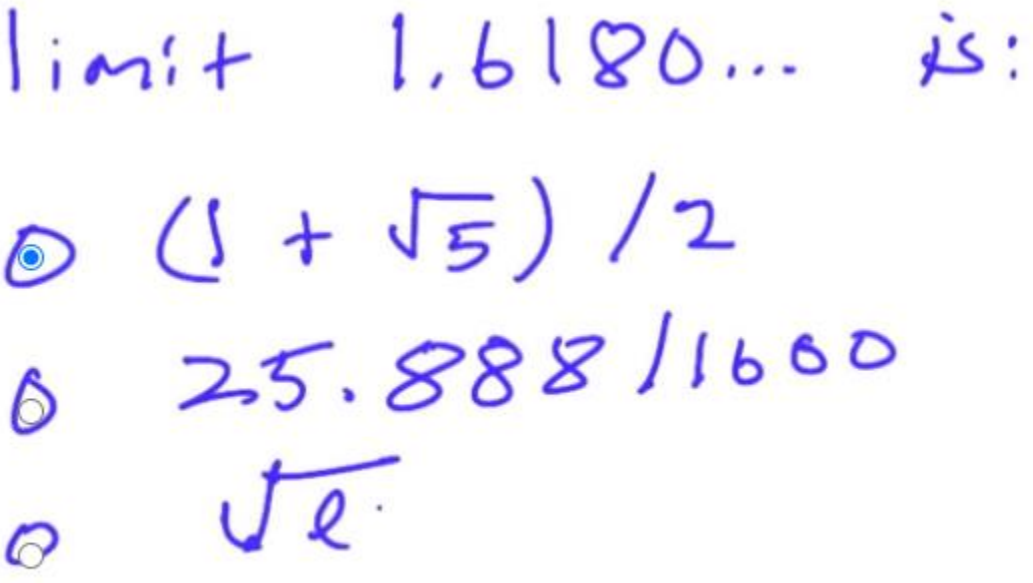
Decorated Decorators

```
def decorator(d):
    "Make function d a decorator: d wraps a function fn."
    def _d(fn):
        return update_wrapper(d(fn), fn)
    update_wrapper(_d, d)
    return _d

## QUIZ: DOES THIS WORK?

def decorator(d):
   "Make function d a decorator: d wraps a function fn. @author Darius Bacon"
   return lambda fn: update_wrapper(d(fn), fn)

decorator = decorator(decorator)

"""       (◉) Yes
          (○) No, results in an error
          (○) No, updates decorator (n_ary) but not decorated (seq)
          (○) No, my brain hurts
```

Cache Management



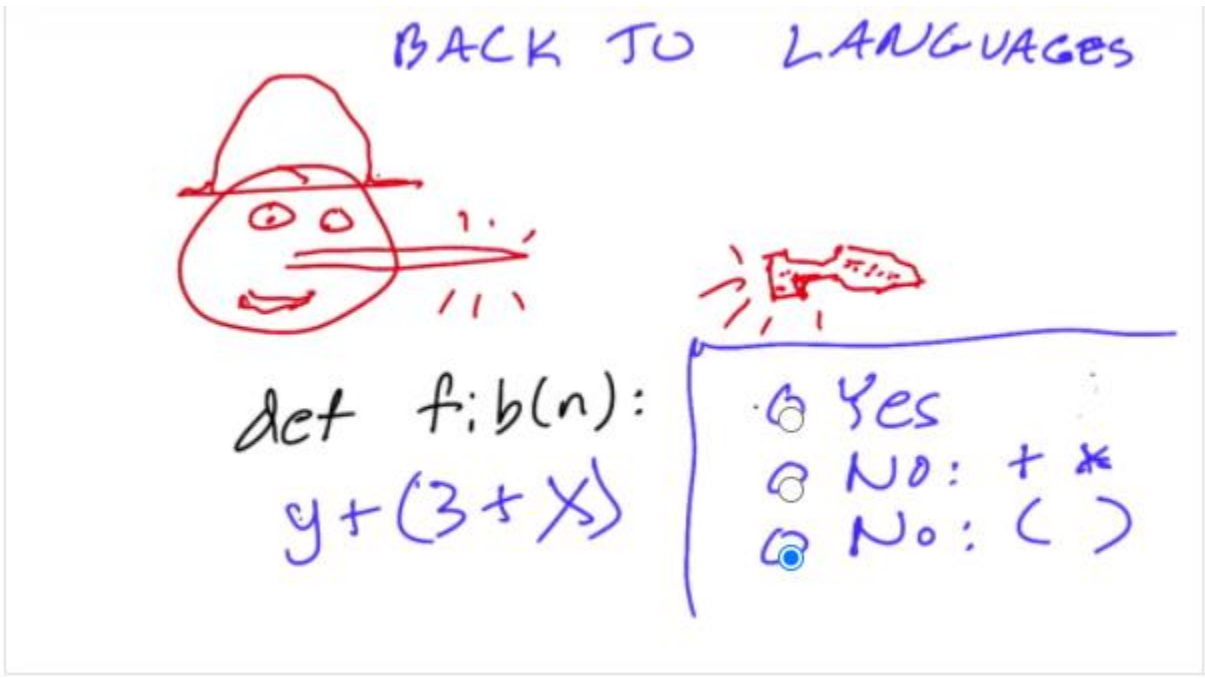
Save Time Now



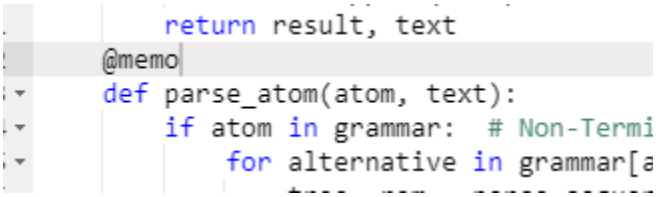
Trace Tool

```
@decorator
def trace(f):
    indent = '    '
    def _f(*args):
        signature = '%s(%s)' % (f.__name__, ', '.join(map(repr, ar
        print '%s--> %s' % (trace.level*indent, signature)
        trace.level += 1
        try:
            result = f(*args)
            print '%s<-- %s == %s' % ((trace.level-1)*indent,
                                      signature, result)
        finally:
            trace.level -= 1
        return result
    trace.level = 0
    return _f
```

Back To Languages



Speedy Parsing



LECCION 10

Json Parser

```
JSON = grammar("""
object => { } | { members }
members => pair , members | pair
pair => string : value
array => [[] []] | [[] elements []]
elements => value , elements | value
value => string | number | object | array | true | false | null
string => "[^"]*"
number => int frac exp | int frac | int exp | int
int => -?[1-9][0-9]*
frac => [.][0-9]+
exp => [eE][-+]?[0-9]+
""", whitespace='\s*')
```

Inverse function

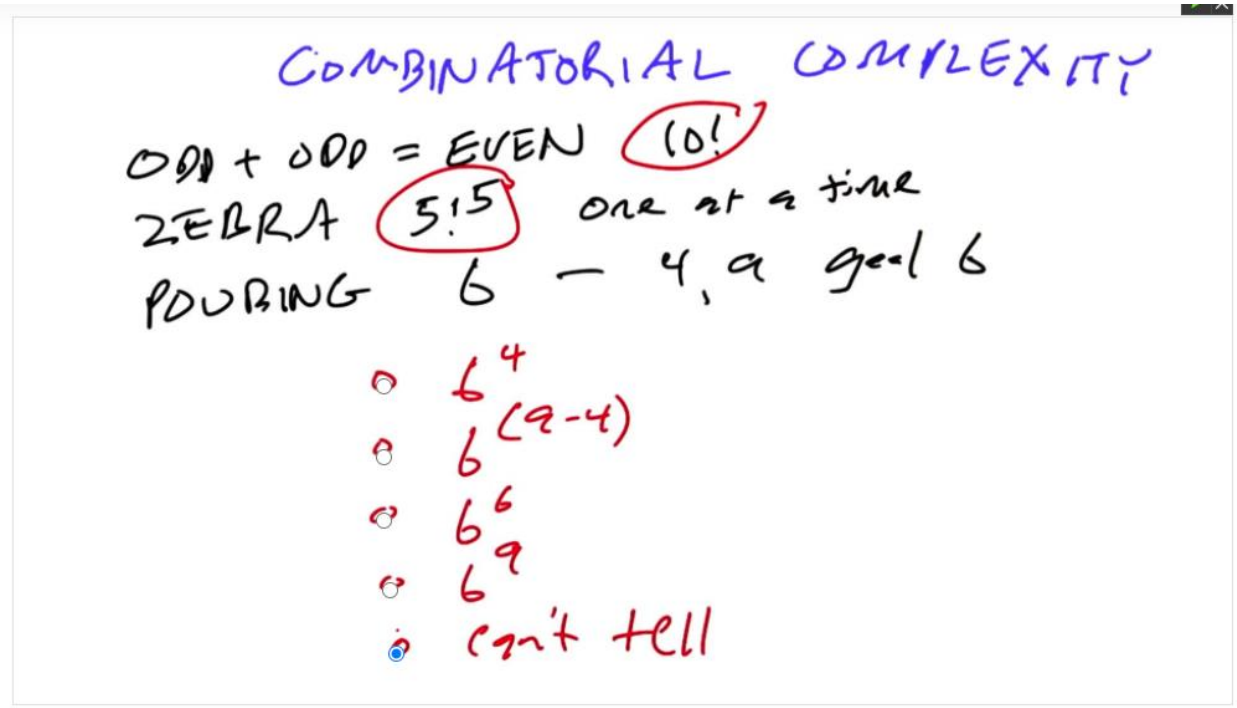
```
def slow_inverse(f, delta=1/1024.):
    """Given a function y = f(x) that is a monotonically increasing
    non-negatve numbers, return the function x = f_1(y) that is an
    inverse, picking the closest value to the inverse, within delta
    def f_1(y):
        x = 0
        while f(x) < y:
            x += delta
        # Now x is too big, x-delta is too small; pick the closest
        return x if (f(x)-y < y-f(x-delta)) else x-delta
    return f_1

def inverse(f, delta = 1/1024.):
    """Given a function y = f(x) that is a monotonically increasing
    non-negatve numbers, return the function x = f_1(y) that is an
    inverse, picking the closest value to the inverse, within delta
    def f_1(y):
        lo, hi = find_bounds(f,y)
        return binary_search(f,y,lo,hi,delta)
    return f_1
```

Find Html Tags

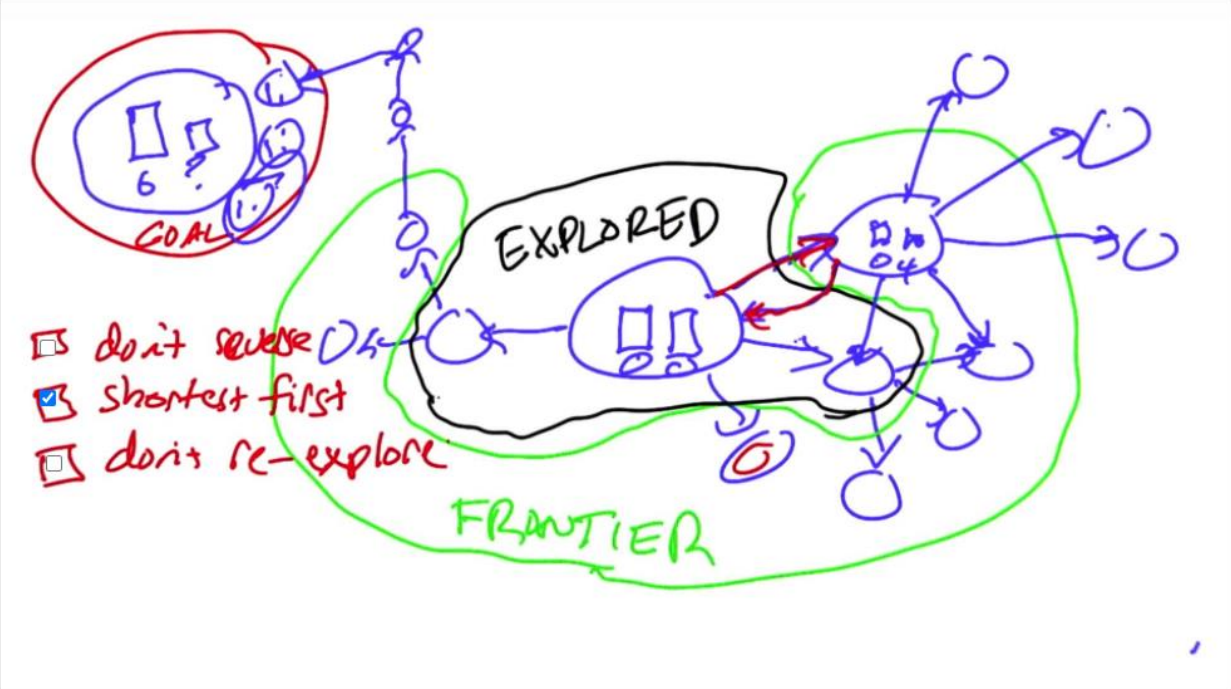
```
8
9  import re
0
1  def findtags(text):
2      parms = '(\w+\s*=s*"[^"]*"\s*)*'
3      tags = '(<\s*\w+\s*' + parms + '\s*/?>)'
4      return re.findall(tags,text)
5
```

LECCION 12

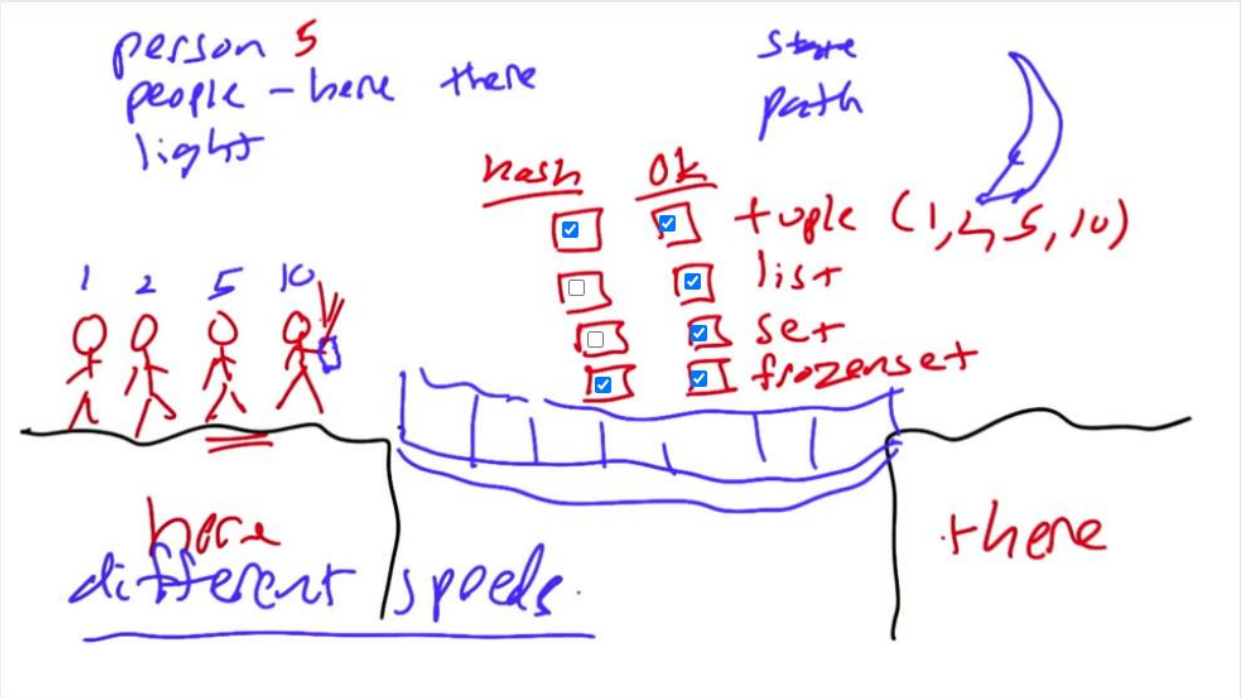
Combinatorial Complexity



Exploring The Space



Representing State



Bridge Successors

```
    if 'light' in here:
        return dict(((here - frozenset([a,b,'light']),
                      there | frozenset([a,b,'light']),
                      t+max(a,b)),
                     (a,b,'->'))
                    for a in here if a is not 'light'
                    for b in here if b is not 'light')
    else:
        return dict(((here - frozenset([a,b,'light']),
                      there | frozenset([a,b,'light']),
                      t+max(a,b)),
                     (a,b,'<-'))
                    for a in here if a is not 'light'
                    for b in here if b is not 'light')

def test():
```

Paths Actions States

```
def path_states(path):
    "Return a list of states in this path."
    return path[0::2]

def path_actions(path):
    "Return a list of actions in this path."
    return path[1::2]
```

Bridge solution

```
def elapsed_time(path):
    return path[-1][2]

print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (5, 2, '->'),
  (frozenset([1, 10]), frozenset(['light', 2, 5]), 5), (1, 1, '<-'),
  (frozenset([1, 10, 'light']), frozenset([2, 5]), 6), (10, 1, '->'),
  (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 16)]

print bridge_problem([1,2,5,10])[1::2]
[(5, 2, '->'), (1, 1, '<-'), (10, 1, '->')]

##      Is that correct?
## ○    Yes
## ◉    No
```

RUN

Debugging

```
def elapsed_time(path):
    return path[-1][2]









print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (5, 2, '->'),
  (frozenset([1, 10]), frozenset(['light', 2, 5]), 5), (1, 1, '<-'),
  (frozenset([1, 10, 'light']), frozenset([2, 5]), 6), (10, 1, '->'),
  (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 16)]

print bridge_problem([1,2,5,10])[1::2]
[(5, 2, '->'), (1, 1, '<-'), (10, 1, '->')]

 ##     Is the program correct now?
 ## ○   Yes
 ## ○   No
 ## ◉   Can't tell
```

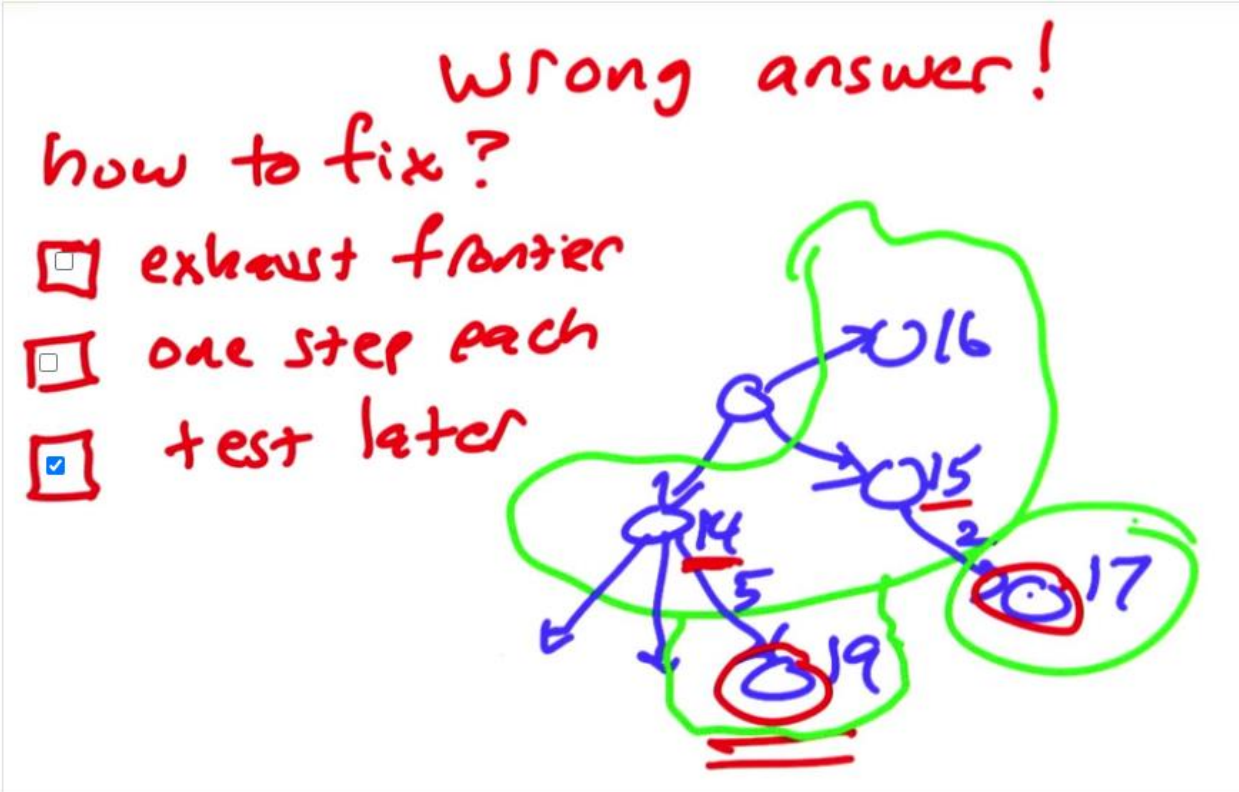
RUN

Did It Work

```
print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (2, 1, '->'),
 (frozenset([10, 5]), frozenset([1, 2, 'light']), 2), (1, 1, '<-'),
 (frozenset([1, 10, 5, 'light']), frozenset([2]), 3), (5, 1, '->'),
 (frozenset([10]), frozenset([1, 2, 5, 'light']), 8), (1, 1, '<-'),
 (frozenset([1, 10, 'light']), frozenset([2, 5]), 9), (10, 1, '->'),
 (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 19)]

print bridge_problem([1,2,5,10])[1::2]
[(2, 1, '->'), (1, 1, '<-'), (5, 1, '->'), (1, 1, '<-'), (10, 1, '->')]

 ##     Is the program correct NOW?
 ## ○   Yes
 ## ○   No, this example is wrong
 ## ◉   No, this example ok, but others wrong
 ## ○   Can't tell
```

RUN

Improving The Solution



Modify Code

```
def bridge_problem(here):
    here = frozenset(here) | frozenset(['light'])
    explored = set() # set of states we have visited
    # State will be a (people-here, people-there, time-elapsed)
    frontier = [ [(here, frozenset(), 0)] ] # ordered list of paths
    while frontier:
        path = frontier.pop(0)
        path = frontier.pop(0)
        here1, there1, t1 = state = path[-1]
        if not here1 or here1 == set(['light']):
            return path
        for (state, action) in bsuccessors(path[-1]).items():
            if state not in explored:
                here, there, t = state
                explored.add(state)
                path2 = path + [action, state]
                path2 = path + [action,state]
                frontier.append(path2)
                frontier.sort(key=elapsed_time)
    return Fail
```

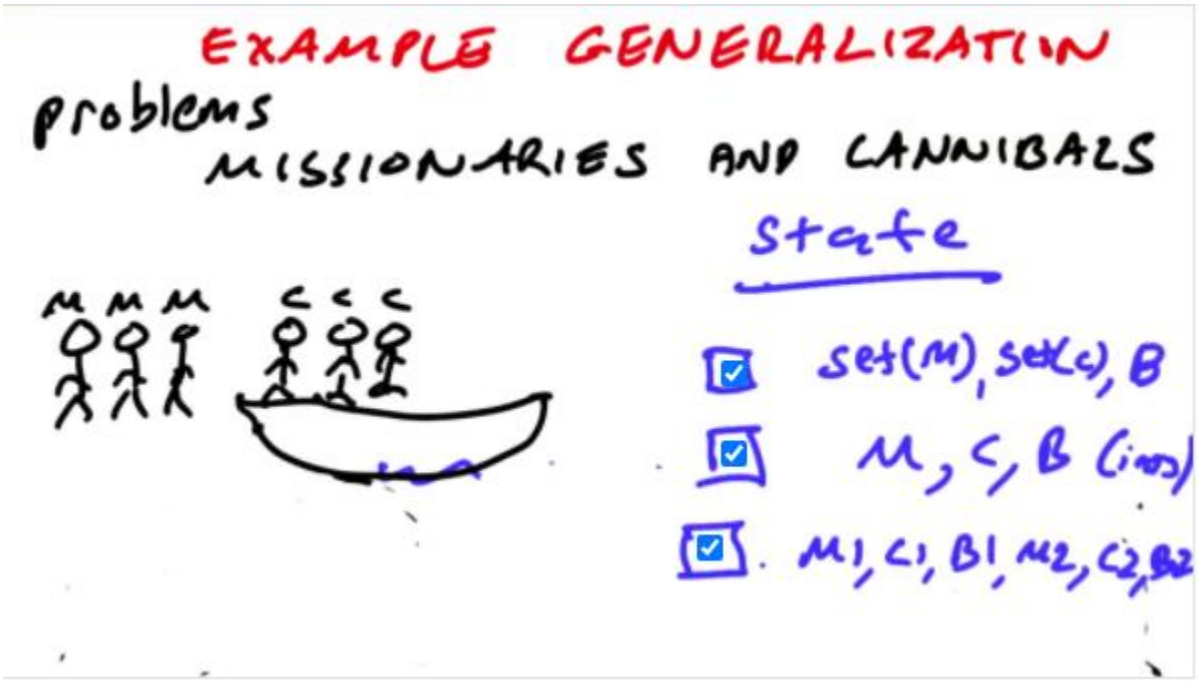
Refactoring Paths

```
def bsuccessors2(state):
    """Return a dict of {state:action} pairs. A state is a
    (here, there) tuple, where here and there are frozensets
    of people (indicated by their travel times) and/or the light."
    here, there = state
    if 'light' in here:
        return dict(((here  - frozenset([a,b, 'light']),
                      there | frozenset([a, b, 'light']),
                      t + max(a, b)),
                     (a, b, '->'))
                    for a in here if a is not 'light'
                    for b in here if b is not 'light')
    else:
        return dict(((here  | frozenset([a,b, 'light']),
                      there - frozenset([a, b, 'light']),
                      t + max(a, b)),
                     (a, b, '<-'))
                    for a in there if a is not 'light'
                    for b in there if b is not 'light')

def bsuccessors(state):
```

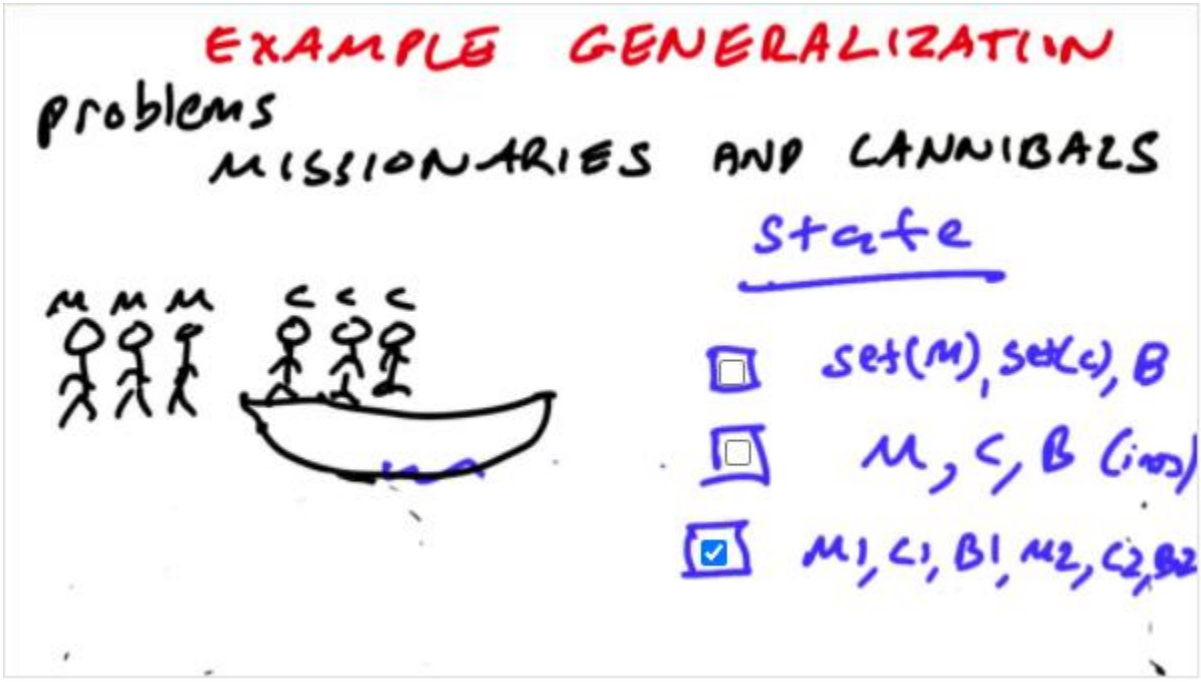
Calculating Costs

```
def path_cost(path):
    """The total cost of a path (which is stored in a tuple
    with the final action."""
    # path = (state, (action, total_cost), state, ... )
    if len(path) < 3:
        return 0
    else:
        action, total_cost = path[-2]
        return toal_cost

def bcost(action):
    """Returns the cost (a number) of an action in the
    bridge problem."""
    # An action is an (a, b, arrow) tuple; a and b are
    # times; arrow is a string.
    a, b, arrow = action
    return max(a,b)
```

Missionaries And Cannibals



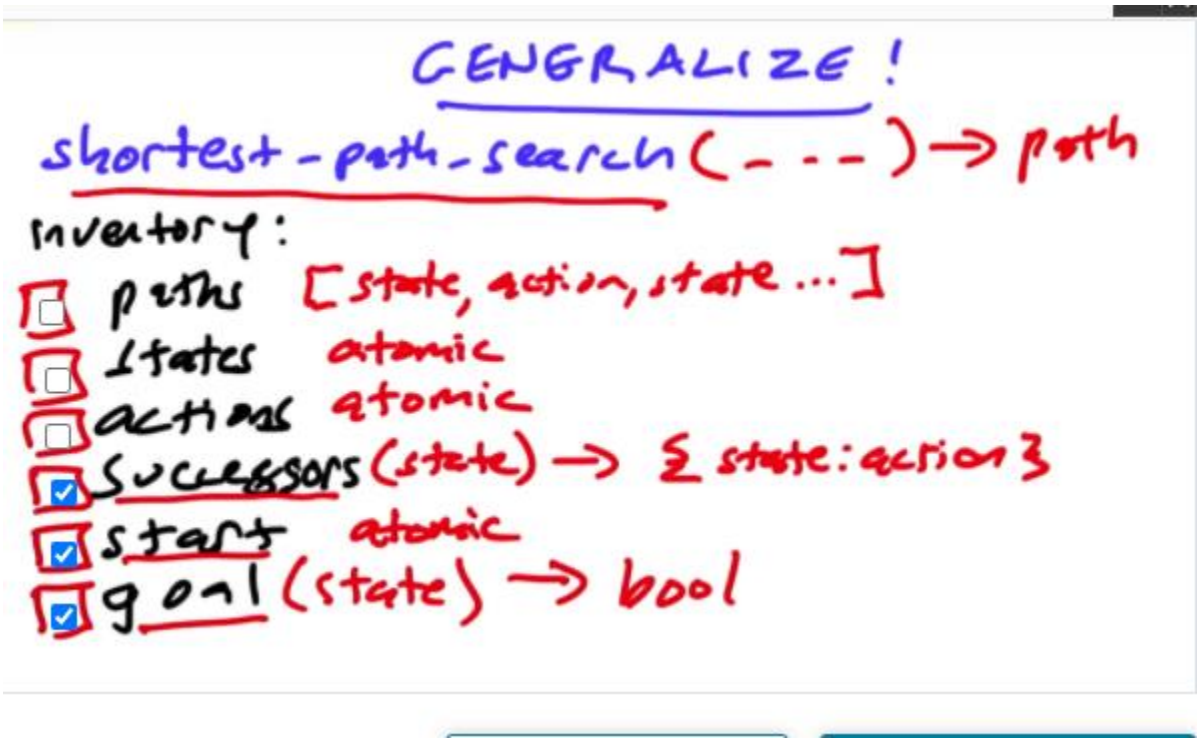
Generalized State



Csuccessors

```
def csuccessors(state):
    """Find successors (including those that result in dining) to
    state. But a state where the cannibals can dine has no success
    M1, C1, B1, M2, C2, B2 = state
    if C1 > M1 > 0 or C2 > M2 > 0:
        return {}
    items = []
    if B1 > 0:
        items += [(sub(state,delta),a+'->')
                  for delta,a in deltas.items()]
    if B2 > 0:
        items += [(sub(state,delta),a+'->')
                  for delta,a in deltas.items()]
    return dict(items)

deltas = {(2,0,1,   -2, 0,-1):'MM',
          (0,2,1,    0,-2,-1):'CC',
          (1,1,1,   -1,-1,-1):'MC',
          (1,0,1,   -1, 0,-1):'M',
          (0,1,1,    0,-1,-1):'C',
```

Shortest Path Search



Sps Function

```
def shortest_path_search(start, successors, is_goal):
    """Find the shortest path from start state to a state
    such that is_goal(state) is true."""
    if is_goal(start):
        return [start]
    explored = set()
    frontier = [[start]]
    while frontier:
        path = frontier.pop(0)
        s = path[-1]
        for (state, action) in successors(s).items():
            if state not in explored:
                explored.add(state)
                path2 = path + [action, state]
                if is_goal(state):
                    return path2
                else:
                    frontier.append(path2)
    return Fail
```

Cleaning Up Mc Problem

```
def mc_problem2(start=(3, 3, 1, 0, 0, 0), goal=None):
    if goal is None:
        goal = (0,0,0) + start[:3]
    return shortest_path_search(start, csuccessors, all_gone) #

def all_gone(state): return state[:3] == (0,0,0)
```

Lowest Cost Search

```
def lowest_cost_search(start, successors, is_goal, action_cost):
    """Return the lowest cost path, starting from start state,
    and considering successors(state) => {state:action,...},
    that ends in a state for which is_goal(state) is true,
    where the cost of a path is the sum of action costs,
    which are given by action_cost(action)."""
    explored = set() # set of states we have visited
    frontier = [ [start] ] # ordered list of paths we have blazed
    while frontier:
        path = frontier.pop(0)
        state1 = final_state(path)
        if is_goal(state1):
            return path
        explored.add(state1)
        pcost = path_cost(path)
        for (state, action) in successors(state1).items():
            if state not in explored:
                total_cost = pcost + action_cost(action)
                path2 = path + [(action, total_cost), state]
                add_to_frontier(frontier, path2)
    return Fail
```

Back To Bridge Problem

```
def bridge_problem3(here):
    """Find the fastest (least elapsed time) path to
    the goal in the bridge problem."""
    start = (frozenset(here) | frozenset(['light']), frozenset())
    return lowest_cost_search(start,bsuccessor2,all_over,bcost) #

def all_over(state):
    here,there = state
    return not here or here == set('light')
```

LECCIÓN 13

Refactoring Bsuccessors

```
def bsuccessors3(state):
    """Return a dict of {state:action} pairs.  State is (here, the
    where here and there are frozen sets of people, light is 0 if
    on the here side and 1 if it is on the there side.
    Action is a tuple (travelers, arrow) where arrow is '->' or '<
    _,_,light = state
    return dict(bsuccessor3(state, set([a,b]))
                for a in state[light]
                for b in state[light])

def bsuccessors3(state, travelers):
    _,_,light = state
    start = state[light] - travelers
    dest = state[1-light] | travelers
    if light == 0:
        return (start, dest, 1), (travelers, '->')
    else:
        return (dest, start, 0), (travelers, '<-')

def test():
```

More Pour Problem

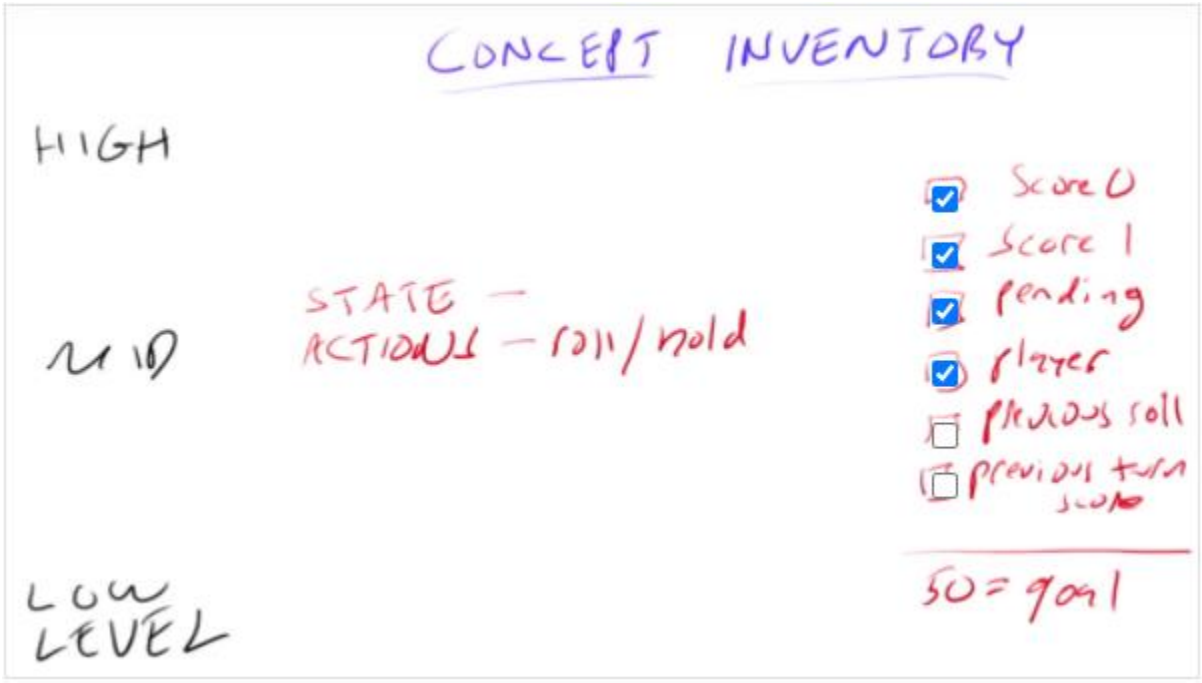
```
    def is_goal(state): return goal in state

    def more_pour_successor(state):
        indices=range(len(state))
        succ={}
        for i in indices:
            succ[replace(state,i,capacities[i])]=('fill,i')
            succ[replace(state,i,0)]=('empty',i)
            for j in indices:
                amount = min(state[i],capacities[j]-state[j])
                state2=replace(state,i,state[i]-amount)
                succ[replace(state2, j, state[j] + amount)]=('pour
            return succ
    if start is None : start = (0,)*len(capacities)
    return shortest_path_search(start,more_pour_successors, is_goa
```

Subway Plannig

```
def subway(**lines):
    """Define a subway map. Input is subway(linename='station1 st
    Convert that and return a dict of the form: {station:{neighbo
    successors = collections.defaultdict(dict)
    for linename,stops in lines.items():
        for a,b in overlapping_pairs(stops.split()):
            successors[a][b]=linename
            successors[b][a]=linename
    return successors

def overlapping_pairs(items):
    return [items[i:i+2] for i in range(len(items)-1)]

boston = subway(
```

LECCION 14

The State Of Pig



Hold And Roll

```
13
14 def hold(state):
15     """Apply the hold action to a state to yield a new state:
16     Reap the 'pending' points and it becomes the other player's turn."""
17     (p,me,you,pending)=state
18     return(other[p],you,me+pending,0)
19
20 def roll(state, d):
21     """Apply the roll action to a state (and a die roll d) to yield a new state:
22     If d is 1, get 1 point (losing any accumulated 'pending' points),
23     and it is the other player's turn. If d > 1, add d to 'pending' points."""
24     (p,me,you,pending)=state
25     if d==1:
26         return(other[p],you,me+1,0)
27     else:
28         return(p,me,you,pendind+d)
29
30 other = {1:0,0:1}
31
```

Clueless

```
possible_moves = ['roll', 'hold']

def clueless(state):
    "A strategy that ignores the state and chooses at random from po
    return random.choice(possible_moves)
```

Hold At Strategy

```
def hold_at(x):
    """Return a strategy that holds if and only if
    pending >= x or player reaches goal."""
    def strategy(state):
        (p,me,you,pending)=state
        return 'hold' if (pending >= x or me + pending >= goal) else 'roll'
    strategy.__name__ = 'hold_at(%d)' % x
    return strategy

goal = 50
def test():
    assert hold_at(30)((1, 29, 15, 20)) == 'roll'
    assert hold_at(30)((1, 29, 15, 21)) == 'hold'
    assert hold_at(15)((0, 2, 30, 10))  == 'roll'
    assert hold_at(15)((0, 2, 30, 15))  == 'hold'
    return 'tests pass'

print test()
```

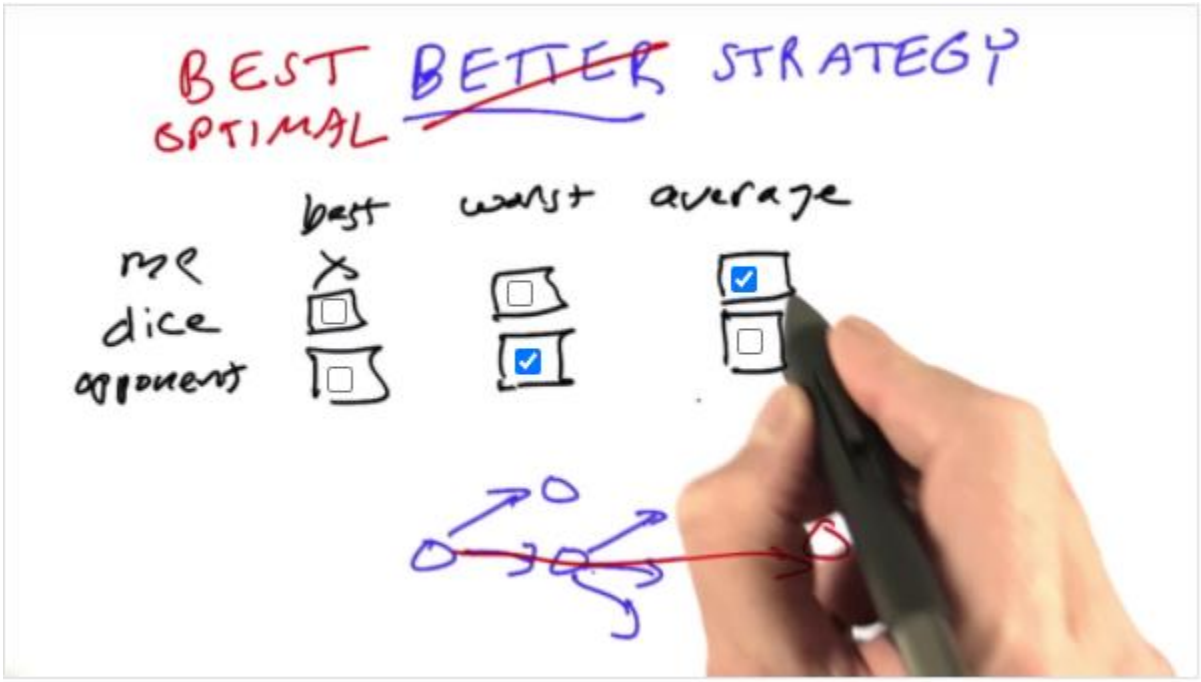
Play Pig

```
def play_pig(A, B):
    """Play a game of pig between two players, represented
    Each time through the main loop we ask the current pla
    which must be 'hold' or 'roll', and we update the stat
    When one player's score exceeds the goal, return that
    strategies=[A,B]
    state=(0,0,0,0)
    while True:
        (p,me,you,pending)=state
        if me >=goal:
            return strategies[p]
        elif you >= goal:
            return strategies[other[p]]
        elif strategies[p](state)=='hold':
            state=hold(state)
        else:
            state=roll(state,random.randint(1,6))
```

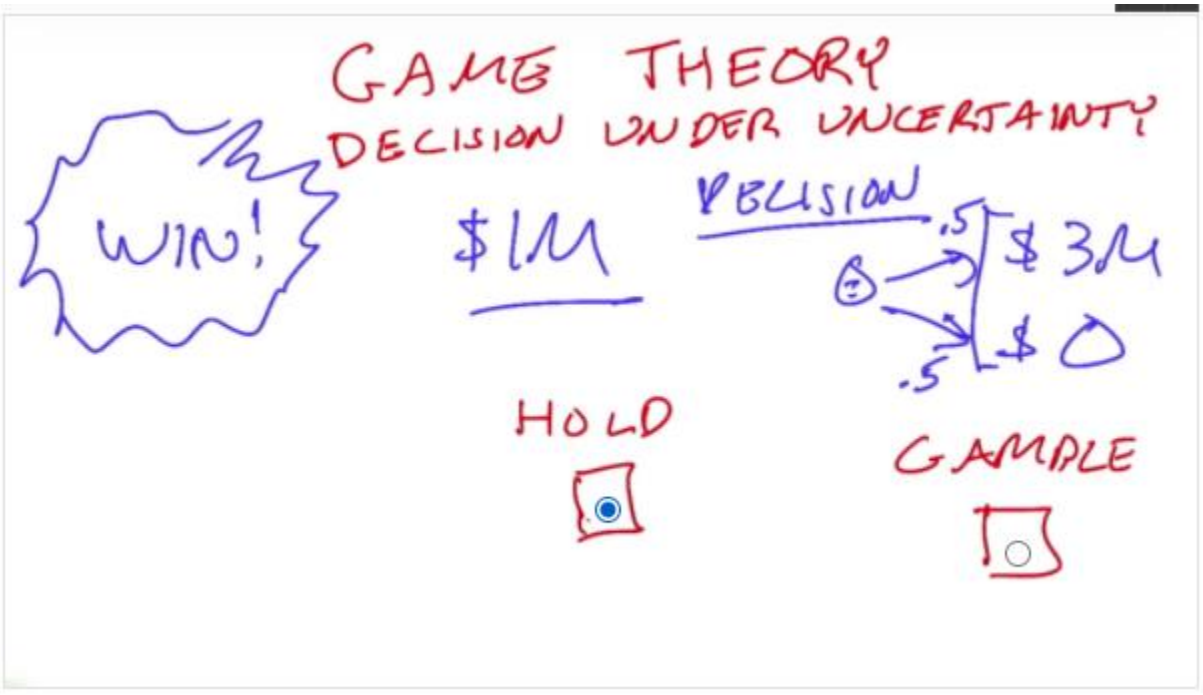
Loading The Dice

```
def play_pig(A, B, dierolls=dierolls()):
    """Play a game of pig between two players, re
    Each time through the main loop we ask the cu
    which must be 'hold' or 'roll', and we update
    When one player's score exceeds the goal, ret
    strategies = [A, B]
    state = (0, 0, 0, 0)
    while True:
        (p, me, you, pending) = state
        if me >= goal:
            return strategies[p]
        elif you >= goal:
            return strategies[other[p]]
        elif strategies[p](state) == 'hold':
            state = hold(state)
        else:
            state = roll(state, next(dierolls))

goal=50

def test():
    A, B = hold_at(50), clueless
    rolls = iter([6,6,6,6,6,6,6,6,2]) # <-- Your
    assert play_pig(A, B, rolls) == A
    return 'test passes'

print test()
```

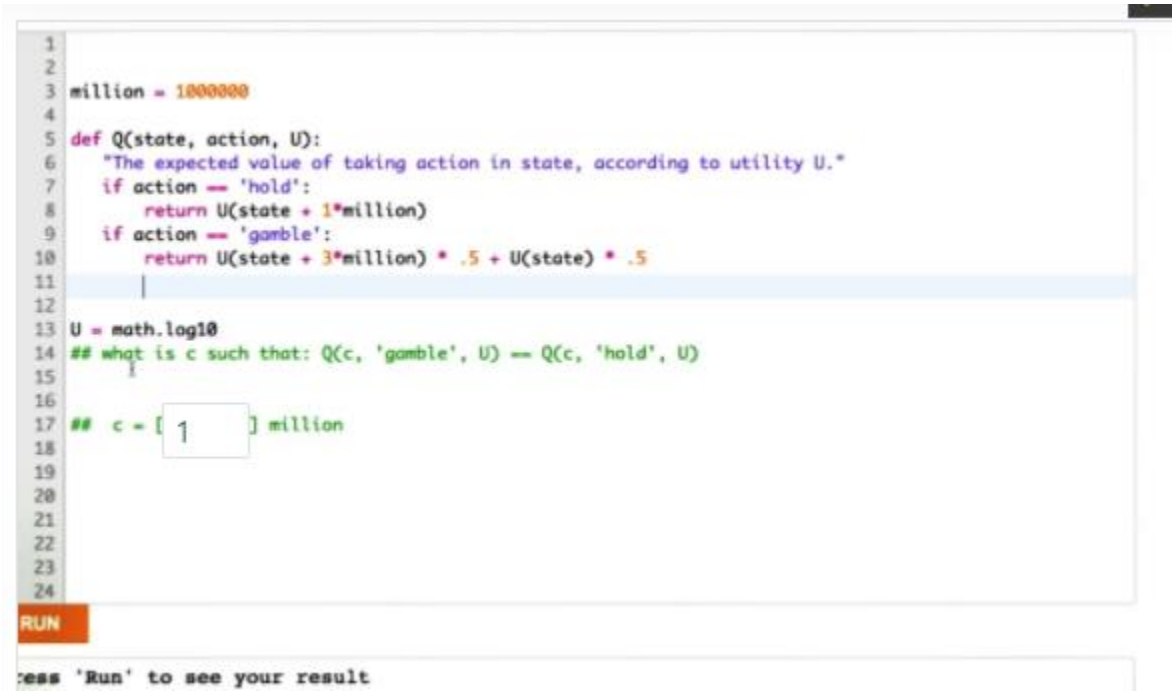
Optimizing Strategy



Game Theory



Break Even Point



```
million = 1000000

def Q(state, action, U):
    "The expected value of taking action in state, according to utility U."
    if action == 'hold':
        return U(state + 1*million)
    if action == 'gamble':
        return U(state + 3*million) * .5 + U(state) * .5

U = math.log10
## what is c such that: Q(c, 'gamble', U) == Q(c, 'hold', U)

##  c = [ 1 ] million
```

RUN

cess 'Run' to see your result

Whats Your Crossover

```
    if action == 'hold':
        return U(state + 1*million)
    if action == 'gamble':
        return U(state + 3*million) * .5 + U(state) * .5


U = math.log10
## what is c such that: Q(c, 'gamble', U) == Q(c, 'hold', U)


##  c = [   1     ] million

c = 1*million
Q(c, 'gamble', math.log10), Q(c, 'hold', math.log10)
(6.301029995663981, 6.301029995663981)


# What's your crossover?  c = [  ]
```

RUN

cess 'Run' to see your result

Maxwins

```
def max_wins(state):
    "The optimal pig strategy chooses an action with the highest
    return best_action(state, pig_actions, Q_pig, Pwin)
```

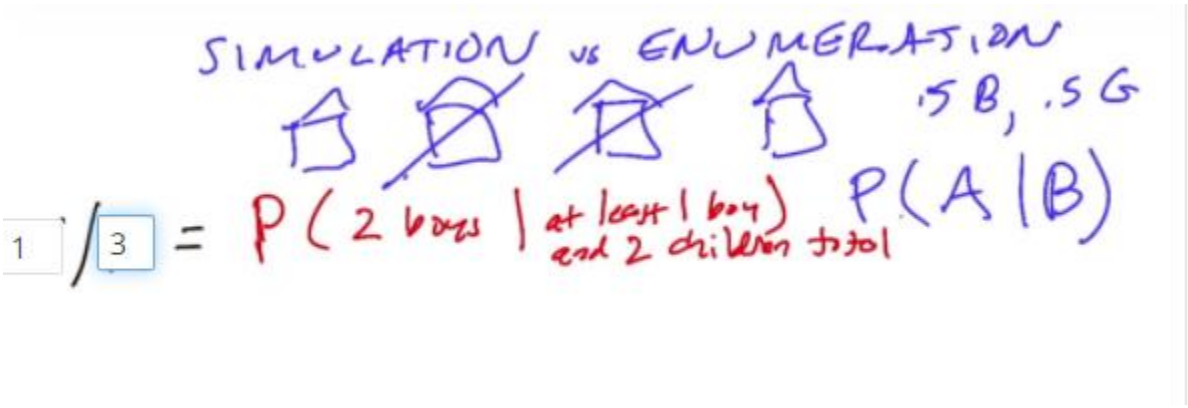
Maximizing Differential

```
def max_diffs(state):
    """A strategy that maximizes the expected difference between
    and my opponent's."""
    return best_action(state,pig_actions,Q_pig,win_diff)
```

Legal Actions

```
def play_pig(A, B, dierolls=dierolls()):
    """Play a game of pig between two players, represented by
    Each time through the main loop we ask the current player
    which must be 'hold' or 'roll', and we update the state ac
    When one player's score exceeds the goal, return that play
    strategies = [A, B]
    state = (0, 0, 0, 0)
    while True:
        (p, me, you, pending) = state
        if me >= goal:
            return strategies[p]
        elif you >= goal:
            return strategies[other[p]]
        else:
            action = strategies[p](state)
            if action == 'hold':
                state=hold(state)
            elif action == 'roll':
                state=roll(state,next(dierolls))
            else:
                return strategies[other[p]]
```

Simulation Vs Enumeration



Tuesday

```
32 what is the probability of two boys?
33 """
34 day = 'SMTWtFs'
35
36 two_kids_bday = product(sex, day, sex, day)
37
38 boy_tuesday = [s for s in two_kids_bday if 'BT' in s]
39
40 print condP(two_boys, boy_tuesday)
41
42 ## Enter as a fraction  [ 13 ]  / [ 27 ]
43
44
45
46
```

LECCIÓN 15

Improving Optimal

```
@memo
def Pwin3(me, you, pending):
   def Pwin3(me, you, pending):
    if me + pending >= goal:
        return 1
    elif you >= goal:
        return 0
    else:
        Proll = (1 - Pwin3(you, me + 1, 0)) + sum(Pwin3(me, you, pending + d) for d in (2,3,4,5,6)) / 6
        return Proll if not pending else max(Proll, 1 - Pwin3(you, me + pending, 0))
```

Doubling Pigs

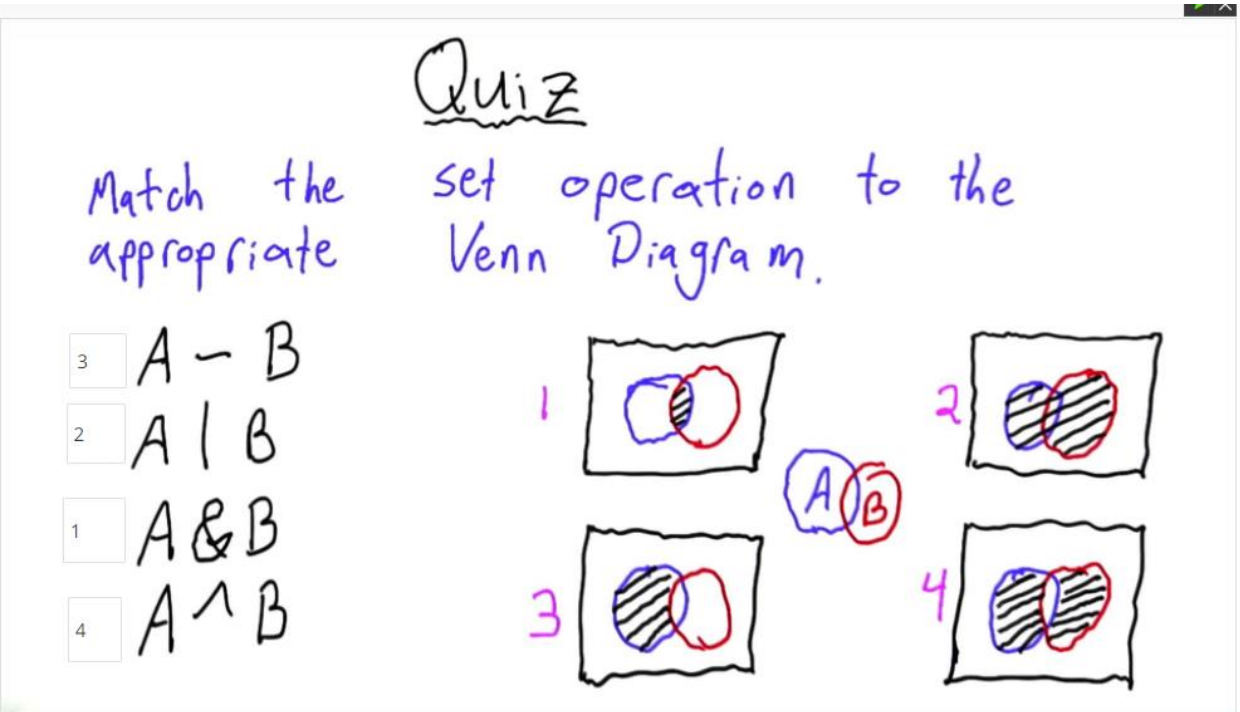
```
def pig_actions_d(state):
    """The legal actions from a state. Usually, ["roll", "hold"].
    Exceptions: If double is "double", can only "accept" or "decline".
    Can't "hold" if pending is 0.
    If double is 1, can "double" (in addition to other moves).
    (If double > 1, cannot "double").
    """
    # state is like before, but with one more component, double,
    # which is 1 or 2 to denote the value of the game, or 'double'
    # for the moment at which one player has doubled and is waiting
    # for the other to accept or decline
    (_, _, _, pending, double) = state
    if double == 'double':
        return ['accept', 'decline']
    actions = ['roll']
    if pending:
        actions.append('hold')
    if double == 1:
        actions.append('double')
    return actions

def strategy_d(state):
    (p, me, you, pending, double) = state
    if 'double' in pig_actions_d(state) and me + pending + 2 >= goal:
        return 'double'
    return hold_20_d(state)
```

Foxes and Hens

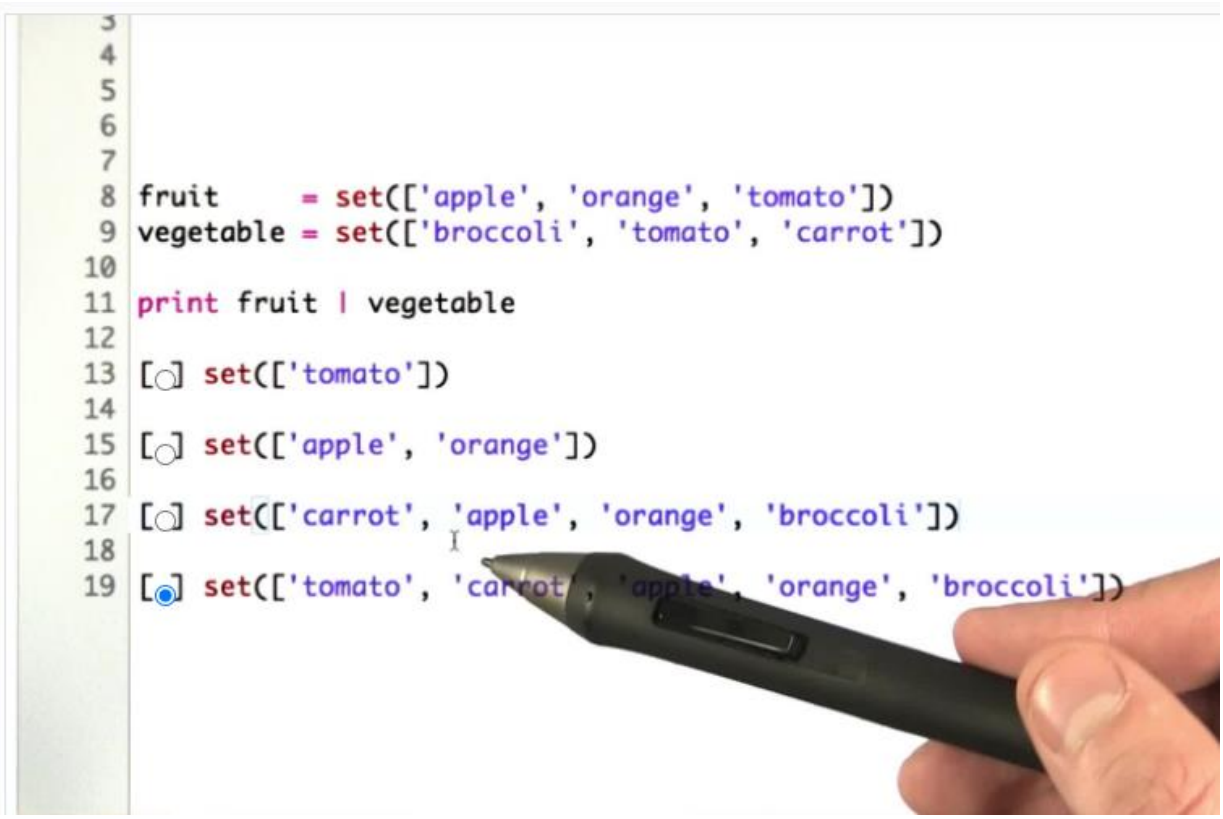
```
def do(action, state):
    "Apply action to state, returning a new state."
    (score,yard,cards)=state
    card=random.choice(cards)
    cards_left=cards.replace(card,'',1)
    if action=='gather':
        return(score+yard,0,cards_left)
    elif action== 'wait' and card=='H':
        return(score,yard+1,cards_left)
    elif action== 'wait' and card=='F':
        return(score,0,cards_left)
    else:
        return state

```

LECCIÓN 17

Set Theory Quiz



Python Sets



Generators

```
5  def sumsquares(n):
6      squares = []
7      for x in range(n+1):
8          squares.append(x*x)
9      return sum(squares)
10
11 print sumsquares(1000000)
12
13 [◉]  Messy: it clutters up our program
14
15 [ ]  Uses memory unnecessarily
16
17 [ ]  Runs too slowly
18
19 [ ]  The code doesn't work!
```

Generators 2

```
5  print sum([x*x for x in range(1000001)])
6
7  [ ] Messy: it clutters up our program
8
9  [◉] Uses memory unnecessarily
10
11 [ ] Runs too slowly
12
13 [ ] The code won't work
```

Generators 3

```
1
4
9
16
25
36
49
64
81
100
>>> def gensquares(n):
...     i = 0
...     while i <= n:
...             yield i*i
...             i += 1
...
>>> gensquares
<function gensquares at 0x1004b17d0>
>>> gensquares(11)
<generator object gensquares at 0x1004a7730>
>>> next(gensquares(11))
0
>>> (•) 0    ( ) 1    ( ) 4    ( ) 121
```

Scope Quiz

```
1  x = 2
2  y = 3
3  def add_nums():
4      y = 6
5      return x + y
6
7  print add_nums()
8
9
10 # what will the output be when I click RUN?
11
12 # (•)   8
13 #
14 # ( )   5
15 #
16 # ( )   NameError
```

LECCIÓN 18

Readwordlist

```
def readwordlist(filename):
    file = open(filename)
    text = file.read().upper()
    wordset = set(word for word in text.splitlines())
    prefixset = set(p for word in wordset for p in prefixes(word))
    return wordset, prefixset
```

Extend Prefix

```
def find_words(letters):
    results = set()

def extend_prefix(w, letters):
    if w in WORDS: results.add(pre)
    if w not in PREFIXES: return
        for L in letters:
            extend_prefix(pre+L,letters.replace(l,'',1),results)
    return results
```

Adding Suffixes

```
90
91 def add_suffixes(hand, pre, results):
92     """Return the set of words that can be formed by extending p
93     if pre in WORDS: results.add(pre)
94     if pre in PREFIXES:
95         for L in hand:
96             add_suffixes(hand.replace(L,'',1), pre+L, results)
97     return results
```

Longest Words

```
71
72 def longest_words(hand, board_letters):
73     "Return all word plays, longest first."
74     words = word_plays(hand, board_letters)
75     return sorted(words, reverse=True, key=len)
76
77
```

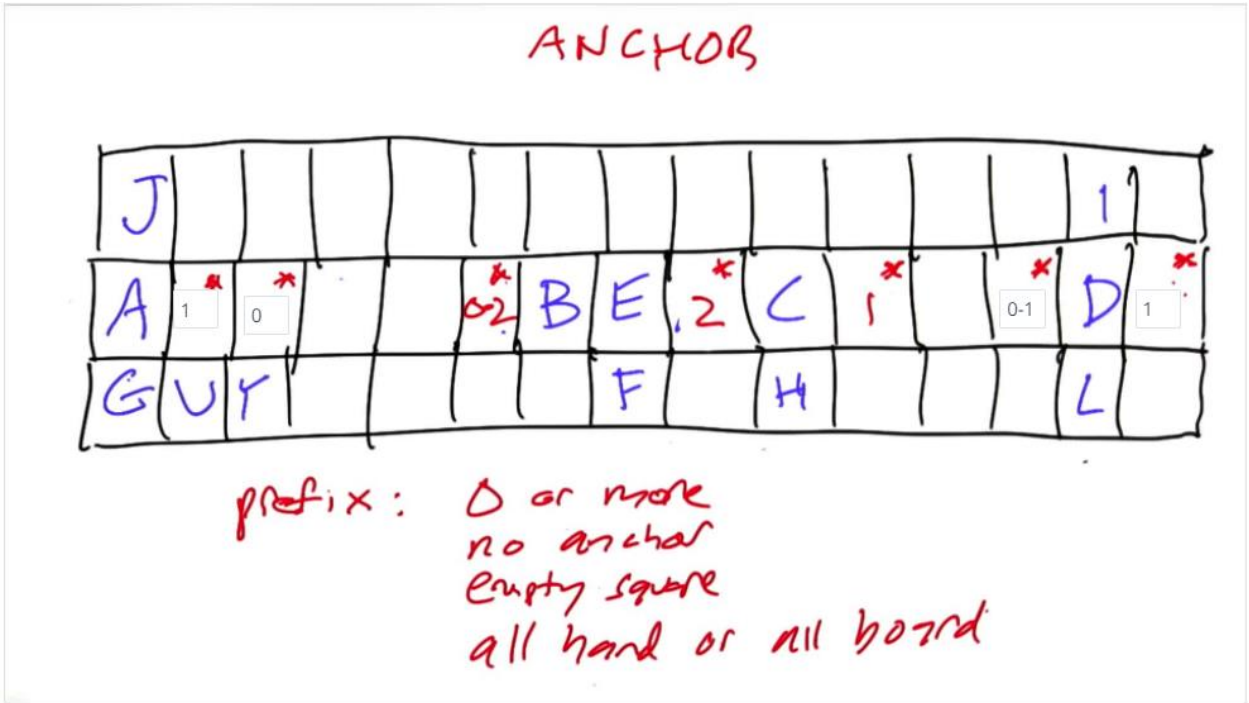
Word Score

```
11
12 def word_score(word):
13     word_score = 0
14     for letter in word:
15         word_score += POINTS[letter]
16     return word_score
17
```

Top N Hands

```
def topn(hand, board_letters, n=10):
    "Return a list of the top n words that hand can play, sorte
    words = word_plays(hand, board_letters)
    return sorted(words,reverse=True, key=word_score)[:n]
```

Anchors



Legal Prefixes

```
    return ('', i-s)

def is_empty(sq):
    "Is this an empty square (no letters, but a valid position on board)."
    return sq  == '.' or sq == '*' or isinstance(sq, anchor)

def is_letter(sq):
    return isinstance(sq, str) and sq in LETTERS

def test_row():
    assert legal_prefix(2, a_row) == ('A', 1)
    assert legal_prefix(3, a_row) == ('',0)
    assert legal_prefix(6, a_row) == ('',2)
    assert legal_prefix(9, a_row) == ('BE',2)
    assert legal_prefix(11, a_row) == ('C',1)
    assert legal_prefix(13, a_row) == ('',1)
    return 'test_row passes'
```

Increasing Efficiency

```
66
67  def find_prefixes(hand, pre='', results=None):
68      """Find all prefixes (of words) that can be made from letters in hand."""
69      global prev_hand, prev_results
70      if hand == prev_hand: return prev_results
71      if results is None: results = set()
72      if pre == '': prev_hand, prev_results = hand, results
73      if pre in WORDS or pre in PREFIXES: results.add(pre)
74      if pre in PREFIXES:
75          for L in hand:
76              find_prefixes(hand.replace(L, '', 1), pre+L, results)
77      return results
```

Show And Spell

```
17
18  def show(board):
19      "Print the board."
20      for i in board:
21          print i
22
```

Horizontal Plays

```
134
135 def horizontal_plays(hand, board):
136     "Find all horizontal plays -- (score, pos, word) pairs -- across all rows."
137     results = set()
138     for (j, row) in enumerate(board[1:-1], 1):
139         set_anchors(row, j, board)
140         for (i, word) in row_plays(hand, row):
141             results.add(((i,j), word))
142     return results
143
```

All Plays

```
151 def all_plays(hand, board):
152     """All plays in both directions. A play is a (score, pos, dir, word) tuple,
153     where pos is an (i, j) pair, and dir is a (delta-_i, delta_j) pair."""
154     hplays = horizontal_plays(hand, board)
155     vplays = horizontal_plays(hand, transpose(board))
156     return (set(((i, j), ACROSS, w) for ( (i, j), w) in hplays) |
157             set(( (i, j), DOWN, w) for ( (j, i), w) in vplays))
158
159
160
```

Making The Board

```
219
220 def show(board):
221     "Print the board."
222     for j, fila in enumerate(board):
223         for i, ss in enumerate(fila):
224             print( ss if (is_letter(ss) or ss == '|') else BONUS[j][i])
225         print
226
```

Making Plays

```
20
21 def make_play(play, board):
22     "Put the word down on the board."
23     (score, (i, j), (di, dj), word) = play
24     for (n, L) in enumerate(word):
25         board[j+ n*dj][i + n*di] = L
26     return board
```

Best Play

```
def best_play(hand, board):
    "Return the highest-scoring play.  Or None."
    plays = all_plays(hand, board)
    return sorted(plays)[-1] if plays else NOPLAY

NOPLAY = None
```

LECCIÓN 19

Anagrams

```
def find_anagrams(letters,previous_word,shortest):
    results=set()
    for w in find_words(letters):
        if len(w)>= shortest and w > previous_word:
            remainder = removed(letters,w)
            if remainder:
                for rest in find_anagrams(remainder,w,shortest):
                    results.add(w+' '+rest)
            else:
                results.add(w)
    return results
```

Blank Tiles

```
# UPDATED: Removed '_' =0 and instead added keys for each lowerc
PUNTOS = dict(A=1, B=3, C=3, D=2, E=1, F=4, G=2, H=4, I=1, J=8,
puntos = defaultdict(int)
for l in ascii_lowercase: puntos[l]
PUNTOS.update(puntos)

def bonus_template(quadrant):
    return mirror(map(mirror, quadrant.split()))

def mirror(secuencia): return secuencia + secuencia[-2::-1]

SCRABBLE = bonus_template("""
'''''''''
```

Boggle

```
def boggle_words(tablero, longitud_minima=3):
    resultados = set()
    filas = size(tablero)
    for (i,sq) in enumerate(tablero):
        if sq != BORDE:
            find_boggle_words(tablero, filas, i, sq, longitud_minima, resultados)
    return resultados

def find_boggle_words(tablero, filas, i, pre, longitud_minima, resultados=None, visitados=set()):
    if resultados is None: resultados = set()
    if pre in PALABRAS and len(pre) >= longitud_minima:
        resultados.add(pre)
    if pre in PREFIJOS:
        ultimo = (i, tablero[i])
        visitados.add(ultimo)
        for nbr in vecinos(i, filas):
            if tablero[nbr] != BORDE and (nbr, tablero[nbr]) not in visitados:
                find_boggle_words(tablero, filas, nbr,
                                  pre + tablero[nbr], longitud_minima, resultados, visitados)
        visitados.remove(ultimo)

    return resultados
```