Design of computer programs Lesson 1!

Learning by doing. There will be a problem and you'll have to give your solution. The instructor will then show and discuss his solution.
There's more than one way to approach a problem and the instructor's solution is not the only/best way.
Solutions are there to help you learn a style and some programming techniques

```
def cs(nums):
    "return sum(x**2 for x in nums)"
```

instructor's solution. as valid & yours if correct but can pick up new things from that

In this class you learn by first debating then try it yourself when the next example comes up.

Write a poker program

?? ① understand problem ② specify spec ③ design code

① hand, card, rank suit ← concept we'll have to deal with
poker(hands) → hand ← best hand

hand rank hand → eg two pair

n-kind ← concepts for hand rank
straight
flush

=

good hand representations

['3S', '3D', '2S', '2C', '7H']
[(1, 'S'), (1, 'D'), (2, 'S'), (2, 'C'), (7, 'H')]

max([3,4,5,0]) → 5
max([3,4,-5,0], key=abs) ← max according to abs
→ 5

=

```
def poker(hands):
    return max(hands, key=hand_rank)
```

↑ corresponds directly to the Problem specification!

```
def hand_rank(hand):
    "to be implemented"
    return None
```

Before writing hand_rank, it's good to think about how the function will be used and to write down some test cases

It's important that each part of the specification gets turned into a code that implements, and a test that tests it

Without test we don't know when we're done, we won't know if we've done it right and we won't have confidence that any future changes might not be breaking something

One important principle of testing is it to test extreme values

poker(hands)

~ with 0, 1, 200 items

the specs do not specify that either. However poker() needs to return one hand! So we make it clear that 0 hands is fine to be an error

poker is passed no more than 1. But nothing in the spec tells it not!

100 would be unusual and it would need a lot of cards, but it can be done

poker has 9 different rankings of the hand, we would want at least one test for each of those to get real confidence we got it right

=

Representing rank 0-8 → rank type

tuples (7,9,5), (7,3,2)
rank 99995 33332
↑ ↑ hands

For the straights

(7,9,5) > (7,3,2) → True

flush (5, [10,8,7,5,3]) ~ full hand

straight (4, 11)
four-high

```
def hand_rank(hand):
    ranks = card_ranks(hand)
    if straight(ranks) and flush(hand):
        return (8, max(ranks))
    elif kind(4, ranks):
        return (7, kind(4, ranks), kind(1, ranks))
```

↑ hand 9,9,9,9,3
(7, 9, 3)

elif
=

To write card_ranks we need to pull out letters from cards (9, K, A) and map them to integers

```
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 ranks
```

=

"AS 2S 3S 4S 5C".split()
our program doesn't recognize it as straight and if it does it returns 5 as highest not 5

We want to modify the function straight, return the correct hand_rank() for this hand, and card_ranks alone to change as well

We can do better than making all the changes above. It seems like we'll really only changing one thing.

In general I want to be able to say that the amount of change should be proportional to the amount of change in the conceptualization. We should be able to confine that change to one place only!

ooo

I want to change card_ranks only

```
def card_ranks(hand):
    ...
    return [5,4,3,2,1] if ranks==
           [14,5,3,4,2] else ranks
```

=

We also want the program to handle ties.

[redacted] would be best to change the poker() function.
hand_rank already return ties

```
poker(hands):
    return max(hands, key=hand_rank)
```

we change max with allmax() that returns a collection of all the maximums

=

n is cards

```
def deal(numhands, n=5, deck=[r+s for r in
    random.shuffle(deck)   '23456789TJQKA'
                           for s in 'SHDC']
return [deck[n*i:n*(i+1)] for i in range
                          (numhands)]
```

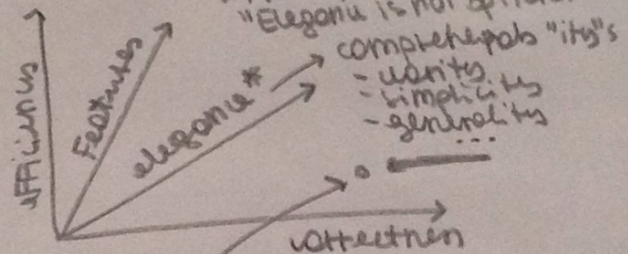
slice of shuffled deck

deck[0:5] n=5, i=0
deck[5,10] n=5, i=1

=

In the real world, problems tend not to ~~to~~ solve just one specific puzzle, but they exist in an environment that evolves over time.

[redacted] program's evolution
*From Richard O'Keefe "Elegance is not optional"
"Elegance is not optional"
complete opts "ity's"
- clarity
- simplicity
- generality
...



our program [redacted] occupies a point on that space. We'd like it to be on the right where the correctness is [redacted] high

over time the program evolves and moves in one of these directions

- IF it's slow we move it up
- move* through the features dimensions if we [redacted] add features
- sometimes we want it to be more elegant, without increasing other dimensions

increasing elegance you may think does nothing - [redacted] it's gaining you something for the FUTure

- easier to maintain
- easier to change
...

* the best is the enemy of the good" Voltaire
IF you're striving for perfection, you may be wasting too much time and you may make a bad engineering tradeoff. Learn to make good tradeoffs: For any change you make in that multidimensional space, there's a benefit, but there's also a cost. It takes you time to do that

Refactoring

Moving along the elegance direction without changing other dimensions is often called refactoring:

We take a program and change it around such that it's cleaner and easier to maintain

```
if kind(3,ranks) and kind(2,ranks):
    return (6, kind(3,ranks), kind(2,ranks))
```

↑
DRY: [redacted] use a different representation

group(7,10,7,9,7) → [redacted]
(3,1,1), (7,10,9)

that makes you realize that [redacted] hand ranks are all the partitions of 5 (5) (4,1) (3,2) (3,1,1) (2,1,1,1) (1,1,1,1,1) in lexicographic order!

[redacted]

```
return (9 if (5,) == counts else
        8 if straight and flush else
        7 if (4,1) == counts else
        6 if (3,2) == counts else
        5 if flush else
        4 if straight else
        3 if (3,1,1) == counts else
        2 if (2,2,1) == counts else
        1 if (2,1,1,1,) == counts else
        0), ranks
```

Summary

- Always start by understanding the problem. Use the specification, see if it makes sense, & if it doesn't consider different ways trying to make sense of it
- Define pieces of a problem. Figure out everything that is relevant with the problem
- As much as you can, try to break the pieces that you have (random, shuffle, map)
- Before you write tests, know what your program does when you specify it both in terms of the pieces that you're defining, and in terms of the tests that tells you what is that this program does
- Explore all dimensions of the design space and decide where we wanna be. Keep moving in the right direction and use good taste to know when to stop

=

computing → sqrt sin → take an input and create a new value as a result

doing → shuffle ↓

take an input and modifies that input

impure Functions = subroutines rather than Functions as they are not Functions in the mathematical sense

input = [3,2,1] ← tests
input.sort()
input == [1,2,3]

PUF Functions

Most of the code is of that type as it's easier to test

sorted([3,2,1])
== [1,2,3]

as we add more state it gonna be more work

Prefered when possible!

testools.combinations(list, n)
↑ takes the combinations of list n at a time

Lesson 2

It's a valuable skill to be able to do quick and dirty calculation. It allow programmers to have the important virtue of [redacted] being lazy.

Learn allows us to [redacted] come up with the simplest design we can, validate through quick and dirty calculations, that that design is efficient or not for what we're trying to do, and then we can stop, we don't have to make it more complex.

This class is about managing complexity, and one of the most important ways to manage complexity is to [redacted] leave it out completely.

IF we can do that, then we're well on our way to better designs

In this class we'll learn how to do that, we'll learn how to do the calculations of when you're efficient enough, we'll learn when to stop and we'll learn how to make programs more efficient

=

Zebra Puzzle

concepts

houses

properties
- nationality
- colors
- pets
- drinks
- smokes

assignment
- locations
- next to
- immed. right
- First middle

assignment

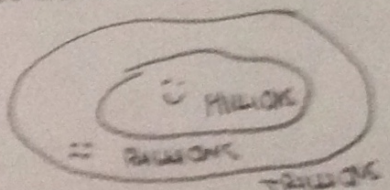
[redacted] ← we're going to try all possibilities

5 properties to 5 houses

5!^5 possibilities ≈ 2 billion

[redacted]

Space of solutions in terms of execution time



=

assignment

red = 1 # house number

↑ use the simplest until you have some proof it won't work

houses = [1,2,3,4,5]

orderings = list(itertools.permutations(houses))

for (red, green, ivory, yellow, blue) in orderings:

for (Englishman, Spaniard, Ukrainian, Japanese, Norwegian) in orderings:

[redacted]

for (dog, snails, Fox, horse, ?=[redacted]) in orderings:

for (Coffee, [redacted], tea, milk, ?) in orderings:

for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:

check [redacted] constrain

5!^5 = 24B → 1000 instructions
2.4 GHz → 100 minutes

Would take 1-3 hours!

...

For: For: For: For: For:
if Englishman == red:

can use is: checks whether two objects are the same object and == checks whether they have the same value. Python handles small integers as a single object

def imright(h1,h2): return h2-h1==1
def next to(h1,h2): return abs(h2-h1)==1

List comprehensions ← generator

[s for r,s in cards] ... more for/if optionally.

term for

result = []
for r,s in cards:
    result.append(s)

[s for r,s in cards if r in 'JQK'] ← generator can guard

result = []
for r,s in cards:
    if r in 'JQK':
        result.append(s)

[s for r,s in cards if ___ for ___ if ___ for ___]
↑ any number of more statements, nested in that order!

generator expressions

g = (term for-clause optional(for/if s)) ← uses () instead of []

- doesn't do any computation
- next(g) loops until finds the first term, and then it stops there
- next(g) again returns the next term and so on ...
- next(g) with last term you get a condition called StopIteration

...

you rarely will be calling next() directly, most of the time you'll be doing this within a for loop

for x2 in (sq(x) for x in range(10) if x%2==0): pass
↑ for loop attempts to call the generator each time, call next() and deal with the StopIteration exception

list((sq(x) for x in range(10) if x%2==0))
↑ can also convert the result. It does all the work and then it returns the result as a list

I never have to deal explicitly with StopIteration

less indentation, stop editing, easier to edit!

The problem with this is that goes through all possible combinations. come combinations, seems silly that we're bothering with them. no

we could more constraints to the earlier time they are defined so that we don't have to bother to go through the remaining loops

for (red,green,ivory,yellow,blue) in orderings
    if imright(green,ivory)
        for (Englishman,Spaniard,Ukranian,Japanese,Norwegian) in orderings
            if Englishmen is red
            if Norwegian is first
            ... so on moving constraint

takes less than 1s !

import time
def t():
    t0 = time.clock()
    zebra_puzzle()
    t1 = time.clock()
    return t1-t0

← took less about 0.4003

You can pass functions on python

def timedcall(fn):
    t0 = time.clock()
    fn()
    t1 = time.clock()
    return t1-t0

allow arguments for fn

def timedcall(fn,*args):
    t0 = time.clock()
    result = fn(*args)
    t1 = time.clock()
    return t1-t0, result

*args can appear at function definition and at function call. In the definition of the function means the function can take any number of arguments and they should be joined up in a tuple called args. In the function call means take that tuple and unpack it and apply it to all those arguments

We can pass the function below as in python functions are first class objects like integers, lists and so on)
We can now decide what happens and when it happens

Timing measurement
one measurement isn't good enough.
There can be errors in measurements. We deal with that by repeating the measurements multiple times

Better measure

def timeit(n,fn,*args):
    times = [timedcall(fn,*args)[0] for i in range(n)]
    return min(times), avg(times), max(times)

...

different timescall() version

If n is int, call n times. [redacted]
else call for n seconds

```
def timedcalls(n, fn, *args).
    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([redacted] times)
```

Another interesting question is how many assignments did we take along the way

A way would be

[redacted]

```
count = 0 [redacted]
for ___
    count += 1
    for ___
        count += 1
        for[redacted] ___
            count += 1
```

In the case of our previous example with assignments for each for loop

Aspects when designing program

- correct → main program
- efficient → efficiency statements
- debug → debugging statements

they will be distinct from the main program

This [redacted] idea is called aspect-oriented programming!

It's an ideal that we can strive for, we [redacted] don't get it completely. But we should always think of whether we can separate those aspects as much as possible

I want to insert counting statements in zebra_puzzle but by keeping the counting separate from the method [redacted] do the minimum amount of damage

```
return next((WATER, ZEBRA)
    for ___ in c(orderings) if ___
    for ___ in c(orderings) if ___ if ___ [redacted]
    for ___ in c(orderings))
```

start counts for each start for

```
def instrument_fn(fn, [redacted] *args).
    c.starts, c.items = 0, 0
    result = fn(*args)
    print '%s got %s with %5d iters over %7d items' % (fn.__name__, result,
        c.starts, c.items)
```

To implement c() we'll use generator functions

```
def ints(start, end):
    i = start
    while i <= end:
        yield i
        i = i + 1
```

makes it a generator function

The function executes, as soon as it sees yield, it generates i, when next() is called it will [redacted] continue from there (yield i) until it [redacted] next yield

L = ints(0, 1000000), L won't be a list right away but a generator

we can yield anywhere in the function.

generator functions allow us to deal with infinite sequences

```
def ints(start, end=None)
...
```

ints(0) → all positive integers

For statements the whole truth!

```
for x in items:
    print x
```

⇒

```
it = iter(items)
try:
    while True:
        x = next(it)
        print x
```

catch the loop and stop

items must be an iterable

- strings
- lists
- tuples
- [redacted] generators

when next stops we get a StopIteration exception [redacted] which is caught here. → [redacted] stop iteration

```
def c(sequence):
    c.starts += 1
    for item in sequence:
        c.items += 1
        yield item
```

sequence starts

First time called

item generates

every successive call will stop at yield

summary

concept inventory
refine ideas
simplest implementation
[redacted]
back envelope (dirty calculations)
refine code

tools → timing, counts
aspects

keep the design as clean as possible, so that we [redacted] can tell what was working on getting the problem right and what on making it more efficient

```
  ODD ← each of these
+ ODD ← letters stands
        for some digit
*EVEN*
```

Figure out which digit stands for which such that the equation will be correct

There are many possibilities but we can make some inferences

eg. *E must be a 1   *N must be even

=

design: write down all rules of arithmetic

complexity → we want a shortcut to eliminate this complexity

=

Try all possibilities: 10! all combinations

Not so many possibilities (10 digits)

design: 'ODD+ODD==EVEN'

- Fill in all permutations for each of the letters

```
ODD + ODD == EVEN
122   122    3435
```

- evaluate, if == true → solution else try another combination

=

concept inventory

- equations – original → str
  – filled → str
- letters — 'D'
- digits — '3'
- assignment letters → digit — table 'D' → '3' str.translate()
- evaluation — eval takes a string and evaluates it as an expression

```
eval('2+7') → 9
eval('2+7==10') → False
```

```
table = string.maketrans('ABC','123')
          ← import string

F = 'A+B==C'
F.translate(table) → '1+2==3'   can eval that

def valid(F):
    try:
        return not re.search(r'\b0[0-9]', F) and eval(F) is True
    except ArithmeticError:
        return False
```

not regular expression

ODD+ODD==EVEN ← we don't usually write numbers that start in 0

re.search(r'\b0[0-9]', F) — looking for 0 followed by 0-9

- r' means the string is a regex
- \b matches at a word boundary
- F ← look for that into F

___ + ___ = (04)05'

matches the regex →

we could come up with errors as well, because of 0~number

012 → 10

the C programming language (in the 1970s) defined that any number starting with 0 is going to be interpreted as an octal

- wrong results 012 → 10
- errors 09 → invalid

```
def fill_in(formula):                                  'I+I=ME'
    letters = ''.join(set(re.findall('[A-Z]', formula)))   'IME'
    for digits in permutations('1234567890', len(letters)):
        table = string.maketrans(letters, ''.join(digits))
        yield formula.translate(table)
```

123, 124, 129, 120, 132, 134, … — no duplicate digits! 10·9·8 → 720 possibilities

generator so that answers are not computed all at once!

from __future__ import division ← import division from python 3

python 2.x	python 3
3/2 → 1	1.5
1/2 → 0	0.5

=

I have yet to convince myself that I've got it right

PROFiling

From terminal

$ python -m cProfile crypt.py

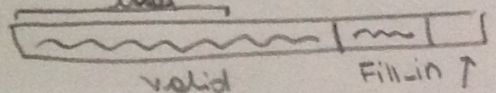
within code

```
import cProfile
cProfile.run('test()')
```

(string to be evaluated)

For our program most of the time is spent in valid. Concentrate efforts where most of the time is

LAW OF DIMINISHING RETURNS



Fill-in won't help much making this bar shorter

within valid(), eval() takes most time. We want to focus on that

eval() is built-in so we can't go editing it to try to make it faster

The only choice then is to make Fewer calls to eval, or to make faster calls

↓

Let's call them Fewer

← divide and conquer

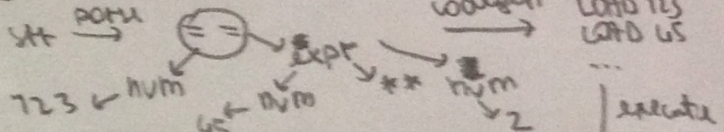
eval('ODD+ODD')
eval('EVEN')

doesn't make much difference here

'YOU==ME**2'

eval('723==45**2')

eval: str → parse



codegen → LOAD 723
LOAD 45
...
execute

False

There's a lot of duplicate work. We do this for every permutation of the digits, but each time we go through, the parsing is going to look exactly the same with the exception of the specific numbers at the bottom. Also codegen will look the same except the number will differ

→ We'd like to do these two parts once and then pass in the specific numbers and then get the result back

Lambda

= creates a function or an expression

anonymous function,

```
def F(Y,O,U,M,E)
    return (100*Y+10*O+U)**
           (10*M+E)**2
```

lambda Y,O,U,M,E : (100* ——

Leave name out

Leave out parentheses

Leave out the return

design:

solve

compile_formula — solves a formula takes str formula and returns a lambda as a result of compiling the formula

compile_word take a word like ME, compile it into 10*M+E and

doesn't change anything that is not an uppercase letter

word[::-1] → reverses a word

enumerate(word) pairs of indexes along with the individual letters

('HOHO') → (0,'H')
(1,'O')
(2,'H')
(3,'O')

re.split('([A-Z]+)', formula)

split up formula by taking sequences of characters A-Z + parenthesis around regex means keep that result

YOU, == ME **2

matched and returned as individual elements by split(), and so on the other parts

['YOU', '==', 'ME', '**2']

tokens = map(compile_word, re.split('([A-Z]+)', formula))

Was faster!

Recap

= python

list comprehensions [x**2 for x in __if__]

generator expressions (__for__ in __if__)

generator functions def f(): ... yield x

handling different types int → float

'polymorphism' timedcalls(n,...)

eval str ⇒ closest function

timing counting time def() timedcall(s)

c() variable running c is only used for debugging as it is not gonna in the final program. Just for debugging purposes

For i in range(__) is who time as i only persists over this short loop

= something lasts longer we probably want it to have a more descriptive name

Regular expression

= find substring in string

s = "some long thing with words"

s.find('word')

return the index at which the word first appears, else -1 if not found

wh regex

'ba a*!' ← any number of the preceding character

*	a*	'', a, aa, aaa ...
?	a?	'', a
.	.	a, b, c, 7, ! ...
^	^b	matches only at the beginning of the line ba, bb ...
$	a$	matches at the end of the line ba ...
''	''	''
a	a	a
ba	ba	ba

search(pattern, text) ← regex in the form of a string

IF pattern occurs in the text return True

match(pattern, text)

= if the pattern appears at the start of the text return True

```
def search(pattern,text):
    if pattern.startswith('^'):
        return match(pattern[1:],text)
    else: return match('.*'+pattern,text)
def match(pattern,text)
    if pattern == '': return True
    elif pattern == '$': return (text == '')
    elif len(pattern) > 1 and pattern in '*?':
        p, op, pattern = 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:]))

def match1(p,text):
    if not text: return F else
    p == '.' or p == text[0]
    return

def match_star(p,pattern,text):
    return match(pattern,text) or (match1(p,text) and
        match_star(p,pattern,text[1:]))
```

= Language

python lang has syntax for statements

expression

format

class - operator overloading

DO MANY SPECIFIC

Regular expressions define a language

a*b*c* describes a language

→ grammar, description of a language ↘ can be difficult to work with.

{a, aa, ab, abb, abc, b} ← language, set of strings

Describe grammar in a more computational format

lit(c)	lit('a')	{'a'}	
seq(x,y)	seq(lit('a'), lit('b'))	{'ab'}	
alt(x,y)	alt(—,—)	{'a','b'}	
star(x)	star(lit('a'))	{'', 'a', 'aa', 'aaa'...}	
oneof(c) ← string of possible characters	oneof('abc')	{'a','b','c'}	
eol ← matches the end of the character string		{''}	
dot	dot	{'a','b','c','d','e'...}	

Instead of returning the match (string), it returns a set of remainders

concept inventory

pattern

text — result

partial result

control over iteration

'a* b*' 'aaab' ← set of remainders

everything else after the match

match1(pattern, text)

→ set of remainders

{'aaab', 'aab', 'ab', 'b'}

= between

'|' means union ← sets in python

- string.startswith(x)
- matches 'a' and 'b' both if a tuple ('a','b') is passed
- any(x==3 for x in l)
  true if l contains a 3

= match(p,t) → '_' | None

new versions → search(p,t) → '_' | None

string

earliest longest match

(except match only matches at the start of the string)

lit(s) ← used to

alt(x,y) ← create

seq(x,y) ← a pattern

↓ pattern created

search(p,—)

match(p,—)

these rely on ... (not part of the programmer's interface)

match_set

(p,t)

↳ {'_','_','_'}

match+remainder = t

Interpreter

match_set(pat, text) → {...}

match_set is an interpreter because it takes the description of the language, namely a pattern, and operates over that pattern

In the match_set implementation, the pattern is defined once, and it's always the same. And yet, every time we get to that pattern we're doing the same process of trying to figure out what operator we have. We should already know that since the pattern didn't change

Another type of interpreter, the compiler which does that work all at once. The very first time when the pattern is defined we do the work of figuring out which parts of the pattern are which, so we don't have to repeat that every time we apply the pattern to the text

A compiler has two steps
- compile(pattern) → c
- c(text)

description of what the compiled code looks like

in our case functions

=

To change our implementation to a compiler

```
def match_set(pattern, text):
    op, x, y = components(pattern)
    if 'lit' == op:
        return set([text[len(x):]]) if text.startswith(x) else null
```

custom object

```
def lit(x): return lambda text: set([text[len(x):]]) if text.startswith(x) else null
```

Function from text to the result that match_set() would have given us

=

We compile into functions but the other other possibilities. Compilers for languages like C generate machine instructions for your computer. Sometimes bytecode instructions for a VM are compiled (Java, Python)

the re.python's bytecode.

```
import dis ; dis.dis(lambda x,y: x+y)
```

=

Compilers have a reputation of being more complex than interpreter, but our compiler here is simpler than the interpreter

=

Recognizer test

match(pat, str) → txt1/None

recognizing whether the prefix of text is in the language defined by pat

generator test

gen(pat) → L ← complete language defined by pat

(a|b)(a|b)

↳ {aa, ab, ba, bb}

languages can be infinite (a*)

- could have a generator function producing one item at a time
- limit sizes of the strings we want

We'll take the compiler approach:
- instead of writing a function generate() we'll have the generator compiled into the pattern

pat({int}) → {str}

↑ compiled function

↑ possible range of lengths that we want to retrieve

... pat = a*

pat({1,2,3}) → {a, aa, aaa}

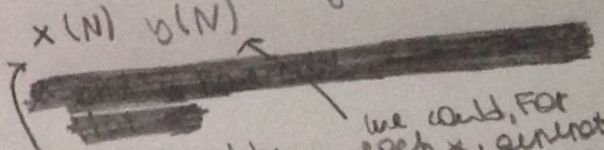
```
def lit(s): return lambda Ns:          ← set of numbers
    set([s]) if len(s) in Ns else null
```

avoiding repetitions

```
def lit(s):
    set_s = set(s)          ← computed once only
    return lambda Ns:
        set_s if len(s) in Ns
        else null
```

genseq

```
def seq(x,y): return lambda Ns:
    genseq(x,y,Ns)
```



N should be anything up to the maximum of Ns

max(Ns) is 10 →

x(N_0..10) y(N_0..10)

a ab
abb bcd
acde
all possible matches for x and y

we could, for each x, generate the ys or generate them all at once and then try to combine them together and see if they match up

abb + ab

→ add up. For each of them. If in Ns, we keep it.

```
def genseq(x,y,Ns):
    Ns = range(max(Ns)+1)
    return set(m1+m2
               for m1 in x(Ns)
               for m2 in y(Ns)
               if len(m1+m2) in Ns)
```

...

the problem with that implementation is that recursive pattern may not return

```
pat = plus(opt('a'))
```

↑ could pick empty string an infinite number of times

we don't want to choose the empty string but we want to make progress and choose something each time through

star() is defined in terms of plus():

```
def star(x): return lambda Ns: opt(plus(x))(Ns)
```

so we only have to fix plus

We have to have some induction where we're reducing something, TO make sure it terminates

Makes sense to reduce Ns

X+ = (X) X* → make sure this X generates at least one character. If we can guarantee that, then when we go to do the X* we've reduced our input (Ns is smaller)

```
def genseq(x,y,Ns, start=0):
    if not Ns: return null
    xmatches = x(set(range(start, max(Ns)+1)))
    Ns_x = set(len(m) for m in xmatches)
    Ns_y = set(n-m for n in Ns for m in Ns_x if n-m>=0)
    ymatches = y(Ns_y)
    return set(m1+m2 for m1 in xmatches
                     for m2 in ymatches
                     if len(m1+m2) in Ns)
```

summary

theory
- pattern grammar → language
- ↑ compiler
- interpreter

practice
- reg exp interpreters
  - expansive
  - natural
- functions
  - composable
  - control over time (can package up computations that we wanna do later)

=

seq(a, seq(b, seq(c,d)))
↳ we want seq(a,b,c,d) REFACTORING

seq is used by different parts of the program that would also need changing

When we consider these changes we want to know if the change is
- backward compatible
- internal / external

changing something within something that doesn't affect the caller

or something on the outside, changing the interface to the rest of the world

```
def seq(x,*args):
    if len(args) == 1
        return ('seq', x, args[0])
    else: ———
```

how seq looks like at
we don't want to repeat ourselves

F2 = n_ary(F)

F2(a,b,c) = F(a,F(b,c))

```
def n_ary(F):
    def n_ary_F(x,*args):
        return x if not args
               else F(x,
                      n_ary_F(*args))
    return n_ary_F
```

```
def seq(x,y): return ('seq',x,y)
seq = n_ary(seq)
```

This pattern is so common in python that there's a special notation for it (decorator notation)

```
@n_ary = function name
def seq(x,y): return('seq',x,y)
```

when I ——— do ———

help(seq) returns docstring and argument list

I get n_ary_F(x,*args) because this is what we returned in n_ary()

TO Fix that:

From functools import update_wrapper

update_wrapper takes two functions and it copies over the function name, the documentation and several other stuff

```
def n_ary(F):
    ...
    update_wrapper(n_ary_F,
                   F)
    return n_ary_F
```

We can make n_ary a decorator and write a definition for it means to be a decorator in terms of update_wrapper()

```
@decorator
def n_ary(f)
    def n_ary(x, *args):
        return x if not args else
                f(x, n_ary_f(*args))
    return n_ary_f
```

```
@decorator
def n_ary          ⇒ n_ary = decorator(n_ary)
```
doc and function name copied over

```
@n_ary
def seq            seq = n_ary(seq)
```
doc and function name copied over ← decorator() does that

```
def decorator(d):
    def _d(Fn):
        return update_wrapper(d(Fn), Fn)
    update_wrapper(_d, d)
    return _d
```

MEMOIZATION

Recap. Sometimes, especially with a recursive function, you'll be making the same function calls over and over again

If the result is always the same and the computation took a long time, it's better to store the result of each value of N with its result in a cache and than look it up each time rather than try to recompute it

```
def Fib(n):
    if n in cache: return cache[n]
    cache[n] = result = ... ← Function body
    return result
```

We can make it a decorator!

```
@decorator
def memo(f):
    cache = {}
    def _f(*args):
        try: return cache[args]
        except KeyError:
            cache[args] = result = f(*args)
            return result
        except TypeError:
            # some element of args can't
            # be a dict key
            return f(args)
    return _f
```

In Python we can either ask for permission or use the try-except pattern to ask for forgiveness afterwards. Here I knew I was gonna need the try anyways for the TypeError (unhashable args)

```
d = {}
d[ls] → KeyError
```

There is no hashcode for a list

lists are mutable and hashcodes might change with them

Types of tools

DEBUG count calls trace → disabled

PERF memo

EXPR n_ary

gives us some idea for the flow of control of the program

```
Fib(6)
-> Fib(6)
   -> Fib(5)
      -> Fib(4)
         -> Fib(3)
            -> Fib(2)
               -> Fib(1)
               <- Fib(1) = 1
               -> Fib(0)
               <- Fib(0) = 1
...
```

```
def disabled(f): return f
```

If I'm using some of these debugging tools, like trace, I can do trace = disabled and reload my program

We don't have to say that disabled() is a decorator because it doesn't create a new function, just returns the original function

BACK TO LANGUAGES

Say we want to parse expressions
We can't use regular expressions because of balanced parentheses

```
y + (3 + x)
```
balanced parentheses are expressed with context-free languages

Grammar → description of the language

Language → set of all possible strings that a grammar describes

```
Exp ⇒ Term [-+] Exp | Term
Term ⇒ [*/]
```

```
G = grammar(r"""
    Exp ⇒ Term Exp | Term
    Term ⇒ Factor [*/]
           Term | Factor
    """)
```

grammar() does this conversion

```
G = {'Exp': (['Term', '[+-]',
             'Exp'], ['Term']),
     'Term': (['Factor',
              '[*/]', 'Term'],
              ['Factor']), }
...
```

Parser ...

parse(symbol, text, G) → (tree, remainder)

Exp → Term [-+] Exp | Term (G must be unambiguous)

(parse left-to-right)

recognizer: Tells us whether the string is part of the language

parser: Tells us whether the string is part of the language and gives a parse tree of the expression

eg: parse('Exp', 'a * x', G)

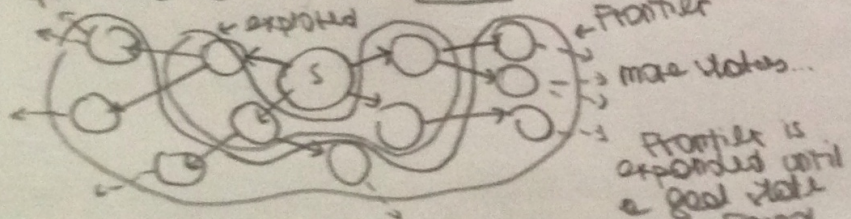
(['Exp', ['Term', ['Factor', ['Var', 'a']]], '*', ['Term', ['Factor', ['Var', 'x']]]], '')

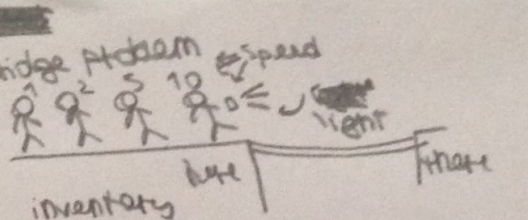
Fail = (None, None) ← NOT parsable return Fail

parse has to deal with
'Exp' symbol to look up in G
regular expression [+-]
alternatives ([...], [...])
list of atoms [..., ..., ...]

Lesson 4 ~
water pouring problem
Inventory glass — allocation
capacity | current ← state is represented with these
goal
pouring — transfer (x → y: y full, x empty)
empty | Fill

this is a combinatorial complexity problem (like cryptarithmetics and zebra)
specifically is a search problem:



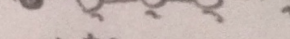
Bridge problem


inventory
person
people - here and there
light
speed

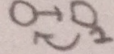
state (here, there, t), elapsed time

frozenset [1,2,5,10,'light'] | frozenset [] ↗

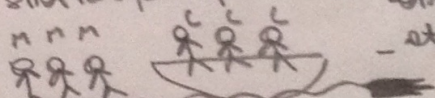
buggy bridge_problem([1,2,5,10])



state (here, there, t) + missing deal with + convention use



Generalize search problems
shortest path problem: missionaries and cannibals



- at most two in a boat
- don't leave more cannibals than missionaries on either side

goal: get everybody across

state
M1, C1, B1, M2, C2, B2
missionaries | cannibals | boats | on other side

Find shortest path from the initial state to the goal state

Let's generalize water pouring and missionaries and cannibals solutions ...

shortest_path_search(start, successors, goal) → path

inventory:
path [state, action, state, ...]
states atomic (specific to the problem)
action atomic
successors(state) → (state, action)
start atomic
goal(state) → bool

doesn't have to know about the state representation

* (start, successors, goal)

lowest cost search, bridge problem

lowest_cost_search(start, successors, goal, cost)

returns the path to goal with lowest cost

summary shortest_path
- search ↗ ↗ least-cost
- different kinds
- subtle → tests
- bugs → tools re-use
- generalize