```
t= [1,2,3,2,1]
reversed(t) → returns function
reversed(t) == t
```

FALSE since one is function and other is list

```
list(reversed(t)) == t
```

True

```
d= {'a':1, 'b':2}
items = iter(d.items())
  next(items) → ('a':1)
```

same thing

```
items = zip(d.keys(), d.values())
next(items) → ('b', 2)
```

Generators

Generator and Generator Functions

```
def plus_minus(x):
    yield x
    yield-x
```

```
t = plus-minus(3)
next(t) → 3
next(t) → -3
```

A generator function is a function that yields values instead of returning them

A normal function returns once; a generator function can yield multiple times

A generator is an iterator created automatically by calling a generator function

When a generator function is called, it returns a generator that iterates over its yields

```
def evens(start, end):
    even = start + (start % 2)
    while even < end:
        yield even
        even += 2
```

```
list(evens(1,10))
[2,4,6,8]
t = evens(2,10)
next(t) → 2
next(t) → 4
next(t) → 6
next(t) → 8
```

Generators & Iterators

Generators can Yield from Iterators

A yield from statement yields all values from an iterator or iterable

```
list(a_then_b([3,4], [5,6]))
```

```
def a_then_b(a,b):
    for x in a:
        yield x
    for y in b:
        yield y
```

```
def a_then_b(a,b):
    yield from a
    yield from b
```

```
def countdown(k):
    if k>0:
        yield k
        yield from
            countdown(k-1)
```

10/07/19: Lecture: Objects

Object-Oriented Programming

A method for organizing Modular Programs
- Abstraction Barriers
- Bundling Together Information and related behavior

A metaphor for computation using distributed state
- each object has its own local state
- each object also knows how to manage its own local state, based on method calls
- method calls are messages passed between objects
- several objects may all be instances of a common type
- different types may relate to each other

Specialized syntax & Vocabulary to support this metaphor

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balances and an account holder; the account class should add those attributes to each newly created instances

```
a = Account('Jim')
a.holder → 'Jim'
a.balance → 0
```

Idea: All bank accounts should have "withdraw" and deposit behaviors that all work the same way

```
a.deposit(15) → 15
a.withdraw(10) → 5
a.balance → 5
a.withdraw(10) →
  'Insufficient funds'
```

Better Idea: All bank accounts share a "withdraw" method and a "deposit" method

Class Statements:

```
class <name>:
    <suite>
```

A class statement creates a new class and binds that class to <name> in the first frame of the current environment.

Assignment & def statement in <suite> create attributes of the class (not names in frames)

```
class Clown:
    nose = 'big and red'
    def dance():
        return 'No thanks'
```

Clown.nose → 'big and red'

Clown.dance() → 'No thanks'

Clown → the physical class

Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

a = Account('Jim')   a.balance → 0

a.holder → 'Jim'

When a class is called:

1. A new instance of that class is created   balance: 0 holder: 'Jim'
2. The _init_ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

```
class Account:
    def _init_(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

a = Account('Jim')   a.balance = 0

b = Account('Jack')   b.holder = 'Jack'

→ every call to Account creates a new Account instance. only 1 Account class

Method

Methods are defined in the suite of a class statement

```
class Account:
    def _init_ (self. account_ holder):
        self. balance= 0
        self. holder = account_holder
    def deposit(self. amount):
        self. balance= self_balance + amount
        return self. balance
    def withdraw (self. amount):
        if amount > self. balance:
            return 'Insufficient
        self. balance = self. balance- amount
        return self. balance
```

These def statements create function objects as always, but their names are bound as attributes of the class

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the objects state

```
class Account:
    def deposit (self, account):
        self. balance = self. balance + amount
        return self. balance
```

2 arguments

* self. whatever
↳ access variables inside the class

Dot notation automatically supplies the first argument to a method.

```
tom_account = Account ('Tom')
tom_ account. deposit (100) ⟶ 100
```

Dot Expressions

Objects review messages via dot notation.

Dot notation accesses attributes of the istance or its class. <expression>. <name>

The <expression> can be any valid Python expression.

The <name> must be a simple name

Evaluate to the value of the attribute looked up by <name> in the object that is the value of the <expression>

Accessing Attributes

Using getattr, we can look up an attribute using a string

```
getattr(tom_account, 'balance') → 18
hasattr(tom_account, 'deposit') → True
```

getattr and dot expressions look up a name in the same way

looking up an attribute name in an object may return:

- one of its instance attributes, or
- one of its attributes of its class

Methods and Functions

Python distinguishes between:

- Functions, which we have been creating since the beginning of the course, and
- Bound Methods, which couple together a function and the object on which the method will be invoked

object + function = bound method

10/9/19: Lecture: Inheritance

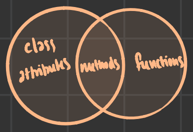
Deadline:

- HW4/lab/HOG comp due Monday
- Ants checkpoint due Tuesday, Early submission for Thursday

Terminology: Attributes, Functions, and Methods

- all objects have attributes, which are name-value pairs
- classes are objects too, so they have attributes
- instance attribute: attribute of an instance
- class attribute: attribute of the class of an instance

Terminology:



Python object system:

- functions are objects
- bound methods are also objects: a function that has its first parameter "self" already bound to an instance
- dot expressions evaluate to bound methods for class attributes that are functions

<instance>.<method_name>

Reminder: Looking Up Attributes by Name

<expression>.<name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Assignment to Attributes

- if object is instance, then assignment sets instance attribute
- if object is class, then assignment sets class attribute

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
tom_account = Account('Tom')
```

```
tom_account.interest = 0.08
```

instance attribute assignment

```
account.interest = 0.04
```

class attribute assignment

```
tom_account.interest → 0.02
Account.interest → 0.04
tom_account.interest → 0.04
```

```
jim_account.interest = 0.08
tom_account.interest → 0.04 (still)
```

Inheritance

- same attributes of parent w/ some different special-case behavior

```
class <name>(<base class>):
    <suite>
```

- "shares attributes", can override inherited characteristics

Inheritance Example

```
class CheckingAccount (Account):
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Looking up Attribute Names

- if in class return attribute value
- otherwise look in base class

10/11/14 - Lecture - Representation

String Representation

String Rep

- An object should behave like kind of data meant to rep
- for instance, by producing string rep of itself
- all objects produce 2 string reps:
  - str - legible to humans
  - repr - legible to Python interp
- often same, sometimes differ

The repr String for an Object

repr returns python expression (string) that evaluates to an equal object

repr(object) → string

12e12 → 12 00000000

print(repr(12e12)) → 12000000000000

repr(min) → < built-in function >

The str string for an object

Human iterable strings:

half = Fraction(1,2)

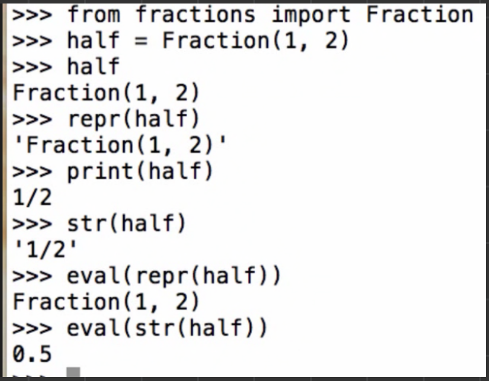
repr(half) → Fraction(1, 2)

str(half) → '1/2'

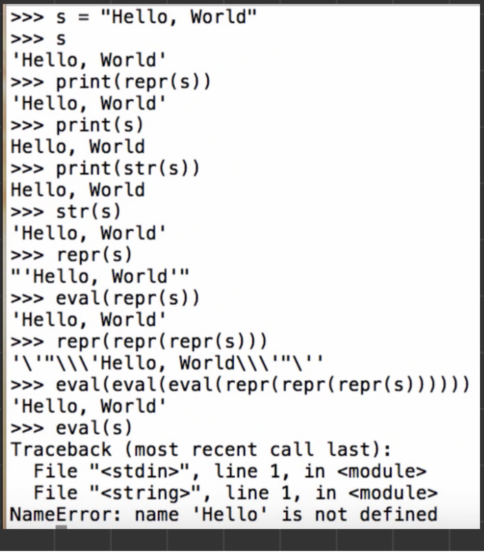
result of calling str on value is what Python prints using print function:

print(half) → 1/2

* can't eval( ) on regular strings and str functions



```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> half
Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> print(half)
1/2
>>> str(half)
'1/2'
>>> eval(repr(half))
Fraction(1, 2)
>>> eval(str(half))
0.5
```



```
>>> s = "Hello, World"
>>> s
'Hello, World'
>>> print(repr(s))
'Hello, World'
>>> print(s)
Hello, World
>>> print(str(s))
Hello, World
>>> str(s)
'Hello, World'
>>> repr(s)
"'Hello, World'"
>>> eval(repr(s))
'Hello, World'
>>> repr(repr(repr(s)))
'\'"\\\'Hello, World\\\'"\''
>>> eval(eval(eval(repr(repr(repr(s))))))
'Hello, World'
>>> eval(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'Hello' is not defined
```

Polymorphic Functions

Polymorphic Functions

poly. func : function that applies to many (poly) different forms (morph) of data

str and repr both polymorphic; apply to any object

repr invokes a zero-argument method __repr__ on its argument

str "

Implementing repr and str

- behavior of repr more complicated than invoking __repr__ on its argument:
- an instance attribute called __repr__ is ignored! only class attributes are found

- behavior of str is also complicated:
- an instance called __str__ is ignored
- if no __str__ attribute is found, uses repr string
- str is a class, not a function

10/14/19 : Lecture : Composition

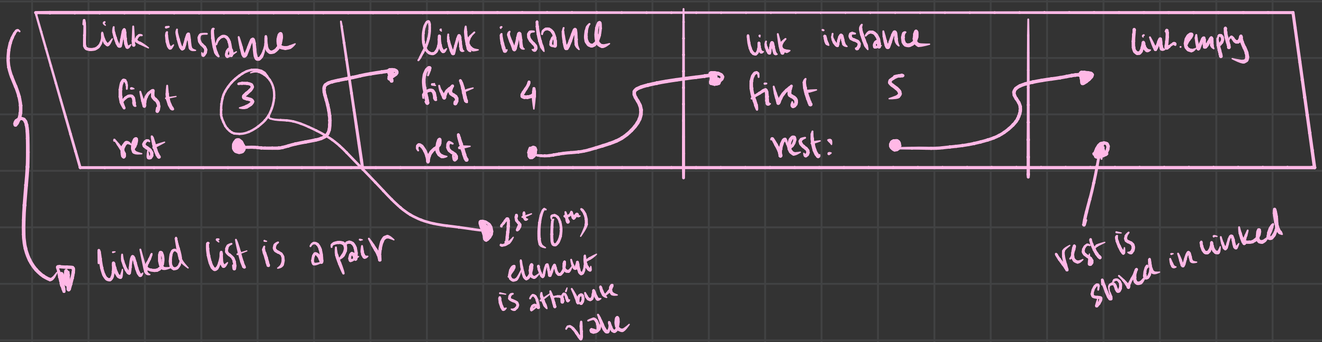
Announcements

- Ants due tmrw & thursday
- HOG due today
- HW & lab today

Linked List

-either empty or consists of first value & rest of linked list

3, 4, 5



link (3, (4, link(5, link.empty)))

evaluate

evaluate

evaluate

Linked List Class

linked list class: attributes passed it _init_

```
class Link:
    def _init_ (self, first, rest=empty):
        assert rest is Link.empty or instance (rest, Link)
        self.first = first
        self.rest = rest
```

returns whether rest is a Link

help(instance): return whether object is an instance or a subclass

s.first → 3

s.rest.first → 4

s.rest.rest.first → 5

4.rest.rest = Link.empty

s → link(6, link(7))

Link(1, Link(Link(2, Link(3)), 4) → < 1 < 2 3 > 4 >

Property Methods

```
@property
@second.setter
```

Tree Class

has a label & list of branches; each branch is a Tree

```
class Tree:
    def _init_ (self, label, branches = []):
—       self.label = label
        for branch in branches:
            assert isinstance (branch, Tree)
—       self.branches = list(branches)
```

all in a class instead of methods!

10/16/19 - Lecture: Efficiency

- do HW5 on a piece of paper for practice
- today is last day of content for midterm 2
- 2 sided sheets for midterm
- no BTree class this sem

Measuring Efficiency

Recursive Computation of the Fib Sequence

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-2) + fib(n-1)
```

fib = count(fib)
fib(5)
fib.call_count → 15

```
def count(f):
    def counted(n):
        counted.call_count += 1
        return f(n)
    counted.call_count = 0
    return counted
```

counts # of times its been called

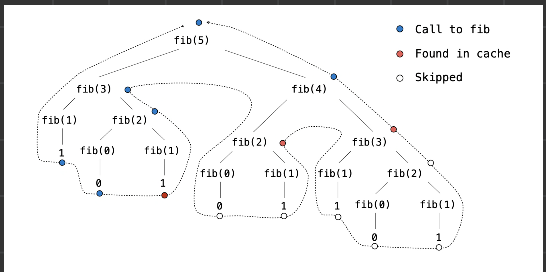
Memoization

Idea: remember the results that have been computed before

```
def memo(f)
    cache = {}  ← empty cache
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

saves in cache

fib = counted(fib)
counted_fib = fib
fib = memo(fib)
fib = count(fib)



Exponentiating

Goal: one more multiplication lets us double the problem size

```
def exp(b,n):
    if n==0:
        return 1
    else:
        return b* exp(b,n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n=0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def exp_fast(b,n):
    if n==0:
        return 1
    elif n%2==0
        return square(exp_fast(b, n//2))
    else:
        return b* exp_fast(b,n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n=0 \\ (b^{1/2 n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

```
def square(x):
    x * x
```

Linear Time:

- doubling input → doubles time

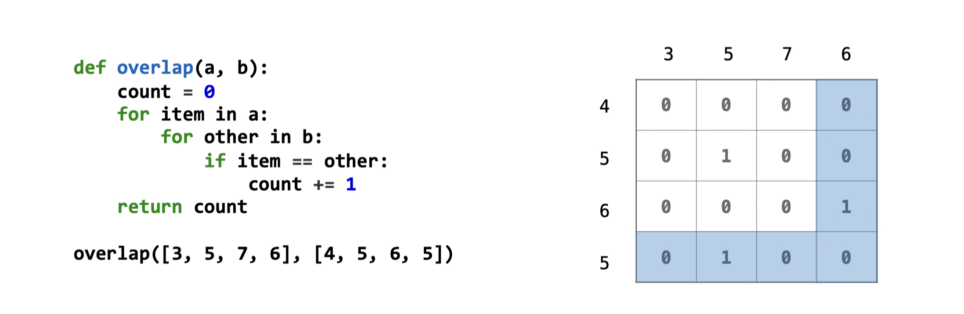
Log Time:

- doubling input increases time by constant C

Orders of Growth

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time



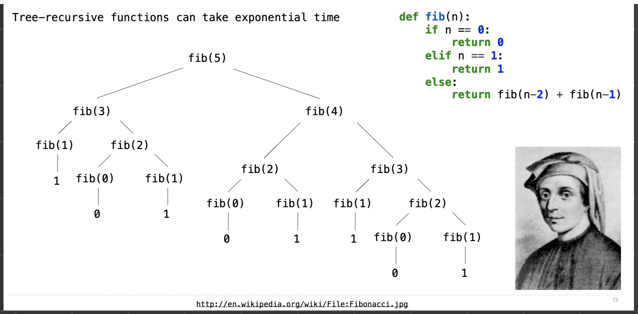
```
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

Exponential Time

Tree-recursive functions can take exponential time



Tree-recursive functions can take exponential time

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

13

Common Orders of Growth

$a \cdot b^{n+1} = (a \cdot b^{n}) \cdot b$

Exponential Growth: recursive fib

incrementing n multiples time by a constant

$a \cdot (n+1)^{2} = (a \cdot n^{2}) + a \cdot (2n+1)$

Quadratic Growth. overlap

incrementing n increases time by n times a constant

$a \cdot (n+1) = (a \cdot n) + a$

Linear Growth slow exp.

incrementing n increases time by a constant

$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$

Logarithmic Growth exp_fast

doubling n only increments time by a constant

Constant Growth Increasing n doesn't affect time

Space and Environments

- Which environment frames do we need to keep during evaluation?
- At any moment there is a set of active environments
- Values and frames in active environment consume memory
- Memory that is used for other values and frames can be recycled

Active Environments

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

Efficiency:

fastest $O(1)$

$O(\log n)$

$O(n)$

$O(n^2)$

...

slowest $O(b^n)$

10/18/19 - Lecture: Decomposition

Modular Design

Separation of Concerns

- A design principle: Isolate different parts of a program that address different concerns
- A modular component can be tested individually

Hog

Hog Game Simulator
- Game Rules
- ordering of Events
- State tracking to determine the winner

Game Commentary
- Event descriptions
- state tracking to generate commentary

Player strategies
- desicion rules
- Strategy parameters

Ants

Ants Game Simulator
- order of actions
- food tracking
- game ending conditions

Actions
- characteristics of different ants & bees

Tunnel Structure
- entrances & exits
- location of insects

Restaurant Search ID:

Restaurant Search Data

n

10/21/19 - Lecture: Review

+= → mutable

_ + _ = → not mutable

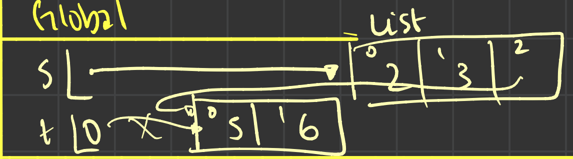
Lists in Environment Diagrams

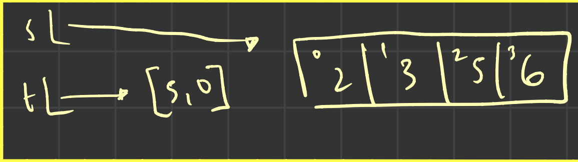
Assume:

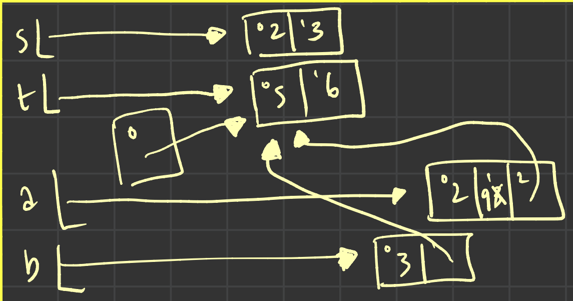
s = [2, 3]

t = [5, 6]

Operation	Example	Result	Global

append - adds one element to a list	s.append(t)
	t=0
		s → [2, 3, [5, 6]]
		t → 0
			

extend - adds all elements in one list to another list	s.extend(t)
	t[1] = 0
		s → [2, 3, 5, 6]
		t → [5, 0]
			

addition and slice create new lists containing existing elements	a = s + [t]
	b = a[1:]
	a[1] = 9
	b[1][1] = 0
		s → [2, 3]
		t → [5, 0]
		a → [2, 9, [5, 0]]
		b → [3, [5, 0]]
			

the list function also creates a new specific list containing existing elements	t = list(s)
	s[1] = 0
		s → [2, 0]
		t → [2, 3]	

slice assignment replaces a slice with new values	s[0:0] = t s[3:] = t t[1] = 0		
pop removes and returns 1st element	t = s.pop()	s → [2] t → 3	

remove removes the first element equal to the argument	t.extend(t)
	t.remove(5)
		s → [2, 3]
		t → [6, 5, 6]	

slice assignment can remove elements from a list by assigning [] to a slice	s[:1] = [] t[0:2] = []	s → [3] t → []	

10/29/19 - Lecture: Scheme

Scheme Fundamentals

- primitive expressions: 2, 3.3, true, +, quotient...
- combinations: (quotient 10 2), (not true)...

numbers are self-evaluating, symbols are bound to values
call expressions include an operator and 0 or more operands in parentheses

```
(quotient 10 2)   >5
(quotient (+ 8 7) 5)   >3
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
>57
```

```
 (+ 1 2 3 4)
>10
 (+)
>0
 (* 1 2 4)
>24
 (*)
>1
 (number? 3)
>#t
 (number? +)
>#f
```

```
> (integer? 2.2)
 #f
>(integer? t)
 #t
```

Special Forms

① evaluate predicate, then consequence/alternative

A combination that is not a call expression is a special form:

- If expression: (if <predicate> <consequence> <alternative>)
- and and or: (and <e_1>...<e_n>) (or <e_1>...<e_n>)
- binding symbols: (define <symbol> <expression>)
- new procedures: (define (<symbol> <formal parameters>) <body>)

```
(define pi 3.14)        pi = 3.14 (assignment)
 (* pi 2)
>6.28
```

```
(define (abs x))      (abs -3)
   (if(< x 0)           >3
      (-x)     }—if less than 0, make -x
      x))        or else just return x
```

```
(define (square x) (* x x))
  > square
```

square x bound to x * x

```
(define (average x y)
        (/ (+ x y) 2)   )
```

average x y bound to x+y/2

```
(define (sqrt x)
   (define (update guess)
      (if =( square guess) x)
         guess
        (update (average guess (/x guess)))))
     (update 1))
```

var

var

runs recursion helper

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4) ))
```

An operator can be a call expression too:

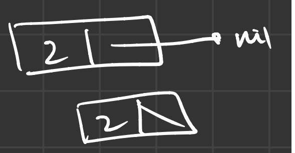
```
((lambda (x y z) (+ x y (square z))) 1 2 3 )
```

evaluates to $x + y + z^2$

Lists

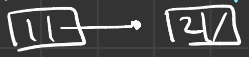
- cons: two-argument procedure that creates a linked list
- car: returns 1st element of list
- cdr: returns rest of list
- nil: the empty list

(cons 2 nil)



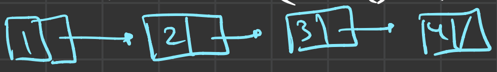
* scheme lists written in parenthesis w/ elements separated by spaces

```
(cons 1 (cons 2 nil))
  > (1 2)
```



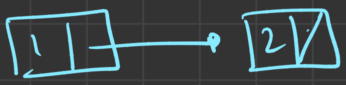
```
(define x (cons 1 (cons 2 nil))
  x
  > (1 2)
```

```
(car x)
>1
(cdr x)
>2
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

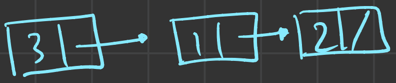


```
(1 2 3 4)
```

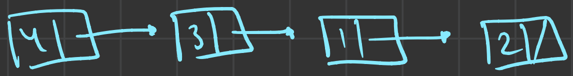
```
(define s (cons 1 (cons 2 nil)))
    > s
s
    > (1 2)
```



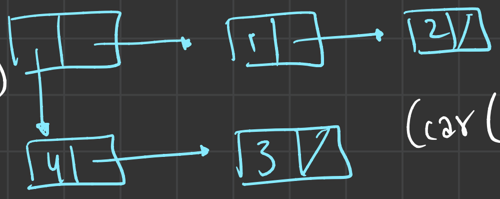
```
(cons 3 s)
 (3 1 2)
```



```
(cons 4 (cons 3 s))
 (4 3 1 2)
```

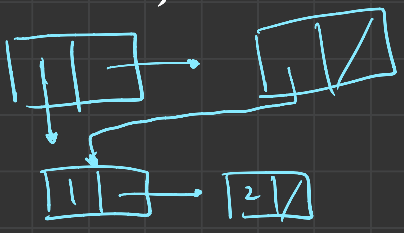


```
(cons (cons 4 (cons 3 n))
((4 3) 1 2)
```



```
(car (cons (cons 4 (cons 3 n)))
     └→ (4 3)
(car (car (cons (cons 4 (cons 3 n))))
    └→ 4
```

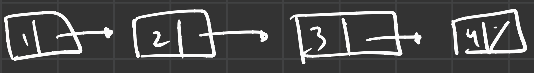
```
(con s (con s nil))
```



```
((1 2) (1 2))

(list? s) #t            (nul? nil) #t
(list? 3) #f            (nul? s) #f
(list? (cas s)) #f
```

```
(list 1 2 3 4)
```



Symbolic Programming

Using text in Scheme: '

```
(list 'a 'b) → (a b)
'(a b c) → (a b c)      or  (quote a)
(car '(abc)) → a
(cdr '(abc)) → (bc)
```

10/30/19: Exceptions

Handling Errors

Sometimes computer programs in non-standard ways

- A function recieves an argument value of improper type
- Some resource is not available
- network connection lost in the middle of data transmission

Exceptions

- Built-in mechanism in a programming language to declare and respond to exceptional conditions
- Python raises exception whenever error occurs
- Exceptions can be handled by the program, preventing the interpreter from halting
- Unhandled exceptions will cause Python to halt execution and print a stack trace

Mastering Exceptions

- Exceptions are objects! They have classes with constructors
- They enable non-local continuations of control:
- If f calls g and g calls h, exceptions can shift control from h to f w/out waiting for g to return

Raise Exceptions

Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the -O flag.

```
python3 -O
```

```
assert False, 'Error'    __debug__ → False    AssertionError False
```

Raise Statements

- Exceptions are raised with a raise statement

```
raise <expression>
```

- <expression> must evaluate to a subclass of BaseException or an instance of one
- Exceptions are constructed like any other object

TypeError - function passed w/ wrong number /argument type

abs('Hello') ; TypeError

NameError - A name wasn't found

'hello' ⟶ NameError, hello is not defined

KeyError - A key wasn't found in a dictionary

{}['hello']

RuntimeError - Catch all for troubles during interpretation

def f(): f() ⟶ RuntimeError

Try Statements

Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

Execution Rule

- The <try suite> is executed first
- If, during the course of executing the <try suite> an exception is raised that is not handled otherwise, and
- If the class of the exception inherits from <exception class>, then
- The <except suite> is executed, with <name> bound to the exception

Handling Exceptions

- Exception handling can prevent a program from terminating

```
try:
    x = 1/0
except ZeroDivisionError as e:
    print ('handling a' type(e))
    x=0
```

Multiple Try statements:

control jumps to except suite of the most recent try statements that handles that type of exception

```
def invert(x)
    y = 1/x
    print ('Never printed if x is 0')
    return y
```

```
def invert_safe(x):
  try:
      return invert(x)
  except ZeroDivisionError as e:
      print ('handled', e)
      return 0
```

10/01/19 - Lecture: Calculator

Announcements:

- Guerilla for tmr
- turn in HW!
- project next week!

Programming Languages

- computer can execute many different languages
- machine language - invoke operations implemented by circuitry of CPU
  - operations refer to hardware memory, no abstraction mechanisms
- High level languages: statements interpreted by another program or compiled into another language
  - provide abstraction, naming, function defining, objects
  - abstract system details to independent hardware

Metalinguistic Abstraction

- define new language tailored to particular type of application or problem domain
- Type of application: Erlang was designed for concurrent programs, has built-in elements for expressing concurrent communication
- Problem domain: MediaWiki mark-up designed for generating static web pages
- Programming Language has:
  - Syntax: legal statements and expressions
  - Semantics: execution/evaluation rule
- To create a new programming langua , need:
  - Specification: document describing precise syntax
  - canonical implementation: interpreter or compiler

Parsing

Reading Scheme Lists

task of parsing together elements creates a string of

Parser

takes text and returns an expression

text → lexical analysis → tokens → syntactical analysis → expression

↓ ↓

- iterative process
- checks malformed tokens
- determines types of tokens
- process one line @ time

- tree recursion
- balances parenthesis
- returns Tree structure
- process multiple lines

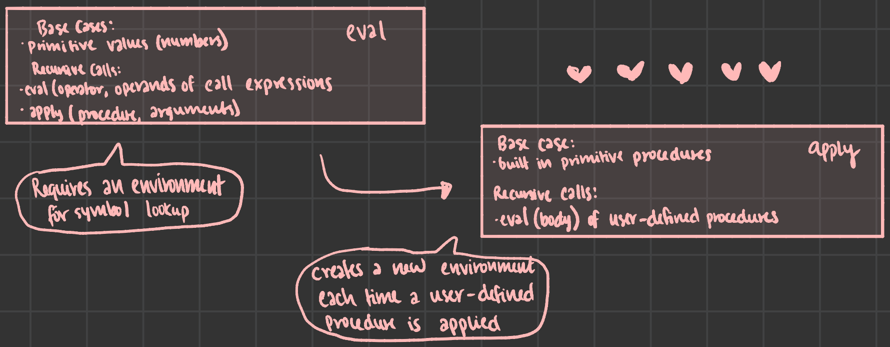
Syntactical Analysis

- identifies hierarchial structure of expression, nested
- each call to scheme_read consumes input tokens

More here!

11/04/19 - Lecture: Interpretors

The Structure of an Interpreter

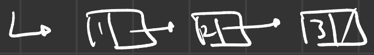

eval
Base Cases:
· primitive values (numbers)
Recursive Calls:
· eval (operator, operands of call expressions
· apply (procedure, arguments)
Requires an environment for symbol lookup
apply
Base case:
· built in primitive procedures
Recursive calls:
· eval (body) of user-defined procedures
creates a new environment each time a user-defined procedure is applied

Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- symbols in enviro
- self-evaluating expressions are returned as values
- all other are represented as Scheme lists, called combinations

```
if <predicate> <consequent> <alternatives>
lambda (<formal-parameters>) <body>)
(define <name> <expression>)
(<operator> <operand 0> ... <operand k>)
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))
(demo (list 1 2))
```



Logical Special Forms

may only evaluate some sub-exp.

if : (if <pred> <cons.> <alt>)

mox to copy

demo

logical special forms

Quotation

quote special form evaluates to quoted expression, not evaluated

(quote <expression>) (quote (+ 1 2)) $\xrightarrow[\text{to}]{\text{evaluates}}$ (+ 1 2)

<expression>

11/06/19 - Lecture: Tail Calls

Dynamic Scope

lexical/static scope - ways names looked up, most typical way; see what name is by inspecting definition

lexical scope: parent of frame is enviro when procedure was defined

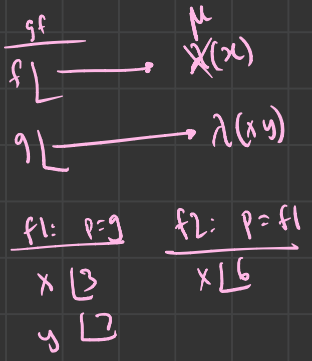
dynamical scope: " " " " " called

(define f (~~lambda~~ mu (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

special form to create dynamically scoped procedures (mu special form only in project 4)

lexical scope: error!

dynamic scope: parent for f's frame is g's frame



Tail Recursion

Functional Programming

- All functions are pure
- No reassignment, no mutable types
- No name bindings permanent
- adv of functional programming
  - value of exp is independent of order
  - sub exp evaluated parallel/demand
  - referential transparency: does not change when we substitute one of its subexp.

no while/for statements

Recursion and Iteration

factorial (n,k) ; computes: n! * k

```
def factorial (n,k):
    if n==0:
        return k
    else: return factorial(n-1, k*n)
```

Time/space
Linear

```
def factorial (n,k):
    while n>0:
        n,k = n-1, k*n
    return k
```

time	space
linear	constant

- scheme is tale recursive!

Tail call

tail call is a call in tail context:

- last body sub-exp in a lambda expression
- sub-exp 2 & 3 in a tail context if expression
- all non-predicate sub exp in tail context cond
- last sub-exp in a tail context and, or, begin, let

```
(define (factorial n k)
   (if (= n 0) k
       (factorial (- n 1)
              (* k n)) ) )
```

last thing you do in a method (compute)

* if tail recursive in scheme, then is linear space

Evaluate with Tail Optimization

11/08/19 - Lecture: Macros

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- primitive expressions: 2 3.3 true + quotient
- combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combination

```
(list 'quotient 10 2)
 (quotient 10 2)
(eval (list 'quotient 10 2))
  5
```

```
 (list + 1 2)       (list '+ 1 2)      (list '+ 1 (+ 2 3))
↳ (#[+] 1 2)      ↳ (+ 1 2)           ↳ (+ 1 5)
```

```
(def (fact n)
   if (= n 0) 1 (* n (fact (- n 1)))))
(define (fact-exp n)
   if (= n 0) 1 (list '* n (fact-exp (- n 1)))))
```

```
(fact 5) → 120
(fact-exp 5) →
 (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in Lisp

Scheme has a define-macro special form that defines a source code transformation

```
(define-macro (twice expr))        (twice (print 2))
   (list 'begin expr expr)           2   2
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions w/out evaluating first
- Evaluate expression returned from the macro procedure

Macro - Crash Course

① Evaluate what you want it to return

② quasiquote everything

③ unquote all the variables and the numbers, keep the function names and arithmetic symbols since you actually want them

*,(car cases)

Macros

```
`(if ,(condition) ,(conseq) (5))
```

* quasiquote everything

if want word → leave alone

if want the variable → or unquote

```
for var in seq (fcn))
`(map ,fcn ,seq)
`( + ,x ,2)
```

└→ would look for variable x)

```
(list 'map fun seq)
```

for x in '(1 2 3) (+ x 1)

map (+ x 1) '(1 2 3)

without macros:

```
(define (twice expr) (list 'begin expr expr))
  (twice '(print 2))

(eval (twice '(print 2)))
    2
    2
```

↳ macros take care of not doing this twice

```
(define-macro .....)

(twice (print 2))
```

For Macro

Define a macro that evaluates an expression for each value in a sequence

```
(define (map fn vals)
   (if (null? vals)
       ()
       (cons (fn (car vals))
             (map fn (cdr vals)))))

(map (lambda (x) (* x x)) '(2 3 4 5))

                    x    (2 3 4 5)  (x * x)
(define-macro (for sym  vals       expr)
  ( list 'map (list 'lambda (list sym) expr) vals ) )

(for x ' (2 3 4 5) (* x x))
```

Quasi-Quotation

` quotes everything

, unquotes part

```
'(abc)
↳ abc
`(abc)
↳ (a b c)
`(a ,b c)
↳ (a 2 c)
```

```
(define expr '(* x x))
   expr
`(lambda (x) ,expr)
 (lambda (x) (* x x))
```

10/13/19 - Lecture: Streams

Order of Growth

Big Theta and Big O Notation for Orders of Growth

- Exponential Growth eg. recursive fib $\Theta(b^n)$

  incrementing n multiplies time by a constant

- Quadratic Growth eg. overlap

  incrementing n increases time by n times a constant $\Theta(n^2)$

- Linear Growth eg. slow exp

  incrementing n increases time by a constant $\Theta(n)$

- Logarithmic Growth eg. slow exp

  incrementing n increases time by a constant $\Theta(n)$

- Constant Growth increasing n doesn't affect time $\Theta(1)$

Efficient Sequence Processing

Sequence Operations

Map, filter, and reduce express sequence manipulation using compact expressions

ex: sum all primes in an interval from a (inclusive) to b (exclusive)

```
def sum_primes(a,b):
    total = 0
    x=a
    while x<b:
        if is_prime(x):
            total = total+x
        x=x+1
    return total
```

space: $\Theta(1)$

```
def sum_primes(a,b):
    return sum(filter(is_prime, range(a,b)))
sum_primes(1, 6)
```

space: $\Theta(1)$

Streams

Streams are Lazy Scheme Lists

A stream is a list, but the rest of the list is computed only when needed:

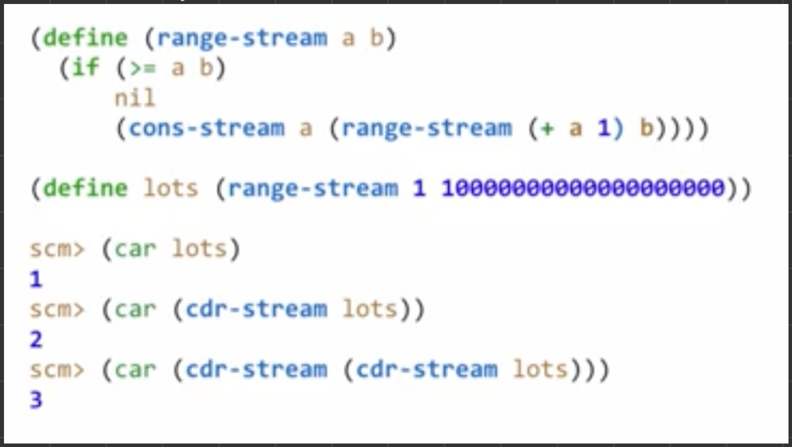
```
(car (cons 1 nil)) → 1              (car (cons-stream 1 nil)) → 1
(cdr (cons 1 nil)) → ()             (cdr-stream (cons-stream 1 nil)) → ()
(cons 1 (cons 2 nil))               (cons-stream 1 (cons-stream 2 nil))
```

Errors only occur when expression is evaluated:

```
(cons 1 (cons (/ 1 0) nil)) → error
(cons-stream 1 (cons-stream (/ 1 0) nil)) → (1 . # (promise not forced))
(car (cons-stream 1 (cons-stream (/ 1 0) nil)) → 1
(cdr-stream (cons-stream 1 (cons-stream (/ 1 0) nil)) → error
```

Streams Ranges are Implicit

A stream can give on-demand access to each element in order



```
(define (range-stream a b)
  (if (>= a b)
      nil
      (cons-stream a (range-stream (+ a 1) b))))

(define lots (range-stream 1 10000000000000000000))

scm> (car lots)
1
scm> (car (cdr-stream lots))
2
scm> (car (cdr-stream (cdr-stream lots)))
3
```

Infinite Stream

Integer Stream

- An integer stream is a stream of consecutive integers
- The rest of the stream is not yet computed when the stream is created

```
(define (int-stream stream)
   (cons-stream start (int-stream (+ start 1))))
```

Recursively Defined Stream

The rest of a constant stream is the constant stream

```
(define ones (cons-stream 1 ones))
```

Combine two streams by separating each into car and cdr

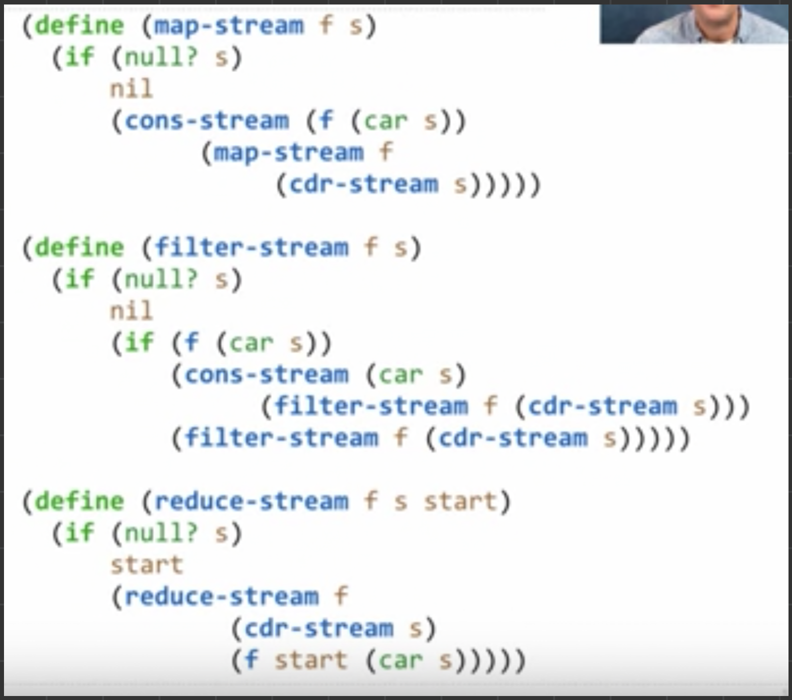
```
(define (add-streams s t)
    (cons-stream (+ (car s) (car t))
        (add-stream (cdr-stream s)
                    (cdr-stream t))))
```

```
(define ints (cons-stream 1 (add-stream ones ints)))
```

Higher-Order Functions

Higher-Order Functions on Streams

implementations are identical, but change cons to cons-stream and cdr to cdr-stream



```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
            (map-stream f
                  (cdr-stream s)))))

(define (filter-stream f s)
  (if (null? s)
      nil
      (if (f (car s))
          (cons-stream (car s)
                (filter-stream f (cdr-stream s)))
          (filter-stream f (cdr-stream s)))))

(define (reduce-stream f s start)
  (if (null? s)
      start
      (reduce-stream f
              (cdr-stream s)
              (f start (car s)))))
```

11/15/19 - Declarative Languages

Declarative Languages

Database Management Systems

Database management systems (DBMS) are important

Table is a collection of records

SQL most widely used, declarative

Declarative Programming

In declarative languages such as SQL & prolog:

- a "program" is a description of the desired result
- interpreter figures out how to generate result

In an imperative language such as Python & Scheme

- a "program" is a description of computational processes
- the interpreter carries out execution/evaluation rules

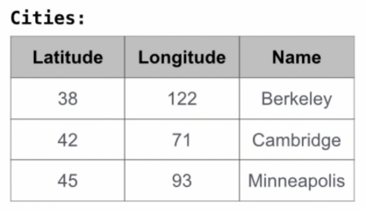
```
create table cities as
   select 38 as latitude, 122 as longitude, "Berkeley" as name union
   select 42,               71,              "Cambridge"          union
   select 45,               93,              "Minneapolis";
```

creating columns

values in column

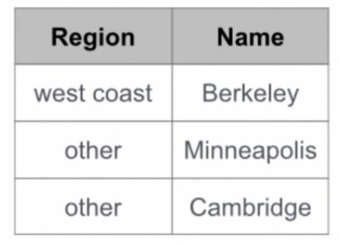
more columns than

```
select "west coast" as region, name from cities where longitude >= 115 union
                                name from cities where longitude < 115
```



Cities:

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis



Region	Name
west coast	Berkeley
other	Minneapolis
other	Cambridge

Structured Query Language (SQL)

SQL overview

SQL language is ANSI and ISO standard, but DBMS

- a select statement creates a new table
- a create table gives global name to a table
- most important action is select statement

Selecting Value Literals

A select statement always includes a comma-separated list of column descriptions

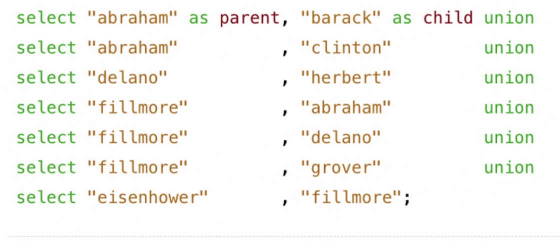
A column description is an expression, optionally followed by as and a column name

select [exp] as [name], [exp] as [name];

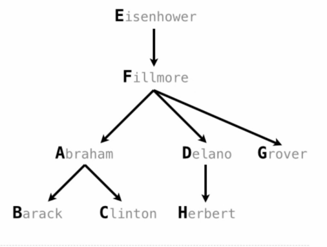
Selecting literals creates a one-row table

The union of 2 select statements is a table containing the rows of both of their results

select "abraham" as parent, "barack" as child union;
select "abraham" , "clinton" union ;,



```
select "abraham" as parent, "barack" as child union
select "abraham"          , "clinton"         union
select "delano"           , "herbert"         union
select "fillmore"         , "abraham"         union
select "fillmore"         , "delano"          union
select "fillmore"         , "grover"          union
select "eisenhower"       , "fillmore";
```



A create table statement gives the result a name

Projecting Table

select Statements Project Existing Tables

A select statement can specify an input table using a from clause

A subset of the rows of the input table can be selected using a where clause

An ordering over the remaining rows can be declared using an order by clause

Column descriptions determine how each input row is projected to a result row

select [exp] as [name], [exp] as name... ← creates table

select [column] from [table] where [cond] order by [order];

select child from parents where parent = "abraham";

selects children column where parent is abraham

select parent from parent where parent > child

select parents from parent table where parent is alphabetically before child

Arithmetic in Select Expressions

In a select expression, column names evaluate to row values

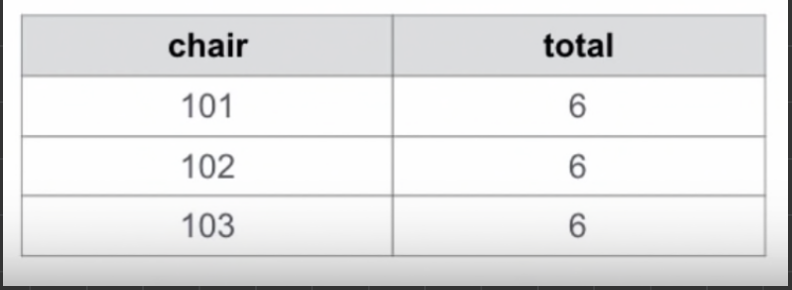
Arithmetic expressions can combine row values and constants

```
create table lift as
    select 101 as chair, 2 as single, 2 as couple  union
    select 102         , 0          , 3            union
    select 103         , 4          , 1 ;

select chair, single + 2 * couple as total from lift;
```

chair
#

total

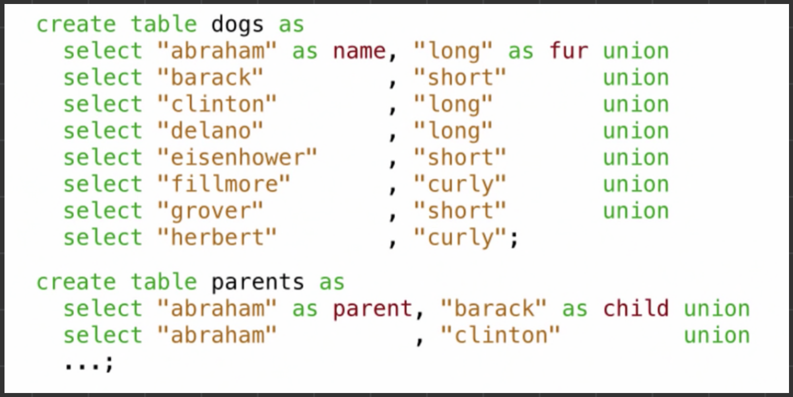


chair	total
101	6
102	6
103	6

11/18/19 - Tables

Joining Tables

Two tables A & B are joined by a comma to yield all combos of a row from A and row from B



```
create table dogs as
  select "abraham" as name, "long" as fur union
  select "barack"         , "short"       union
  select "clinton"        , "long"        union
  select "delano"         , "long"        union
  select "eisenhower"     , "short"       union
  select "fillmore"       , "curly"       union
  select "grover"         , "short"       union
  select "herbert"        , "curly";

create table parents as
  select "abraham" as parent, "barack" as child union
  select "abraham"          , "clinton"         union
  ...;
```

Select the parents of curly-furred dogs

```
select parent from parents, dogs
  where child = name and fur = "curly";
```

makes table w/ all combos of 2 table rows joined

wants parent row only

their children select the names

only ones that are curly

```
select * from parents, dogs
  where child = name;
```

*only rows of table where name of one is child of other (basically group)

Dot Expressions and Aliases

Joining Table with Itself

```
select a.child as first, b.child as second
  from parents as a, parents as b
where a.parent = b.parent and a.child < b.child
```

first	second
children	children

siblings

Joining Multiple Tables

Multiple Tables can be joined to yield all combos of rows from each

```
create table grandparents as
    select a.parent as grandog, b.child as granpup
        from parents as a, parents as b
    where b.parent = a.child
```

select all grandparents w/ same fur as grandchildren

```
select grandog from grandparents, dog as c, dog as b
    where   c. name = grandog   and
            d. name = grandpup   and
            c. fur = d. fur
```

Numerical Expression

Expressions can contain function calls and arithmetic operators

```
[exp] as [name], [expression] as [name], ...
select [colums] from [table] where [expression] order by [expression];
```

combine values: +, -, *, /, %, and, or

transform values: abs, round, not, -

compare values: <, <=, >, >=, <>, !=, =

```
create table cold as
    select name from cities where latitude >= 43;

create table distances as
    select a.name as first, b.name as second,
        60 * (b.latitude - a.latitude) as distance
        from cities as a, cities as b;
```

String Expressions

🟢 String Values can be combined to form longer strings

```
select "hello," || " world";
 hello, world
```

🟡 Basic string manipulation is built in SQL

```
create table phrase as select "hello, world" as s;
select substr(s, 4, 2) || substr (s, intr (s, " ")+1, 1) from phase;
```

🔴 Strings can be used to represent structured values,

```
create table lists as select "one" as car "two, three, ur" as cdr;
select substr (cctr, 1, instr (cdr, ", ") -1) as cadr from lists;
```

11/20/19- Aggregation

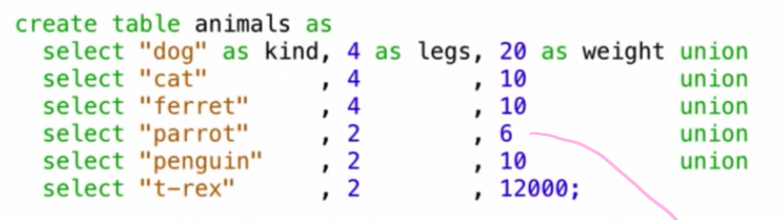
Aggregate Functions

select [columns] from [table] where [expression] order by [expression];

v

[expression] as [name], [expression] as [name], ...

An aggregate function in the [columns] clause computes a value from a group of rows



```
create table animals as
  select "dog" as kind, 4 as legs, 20 as weight union
  select "cat"        , 4       , 10          union
  select "ferret"     , 4       , 10          union
  select "parrot"     , 2       , 6           union
  select "penguin"    , 2       , 10          union
  select "t-rex"      , 2       , 12000;
```

max(legs)
4

select max(legs) from animals;
4

select max (legs-weight) + 5 from animals;
1

select max(legs), min(weight) from animals;
4 | 6

select max(weight) - min(legs) from animals;
-2

select min(legs), max(weight) from animals
where kind <> 'trex'

20

select avg(legs) from animals;
3.0

select count(*) from animals;
6

select count (distinct legs) from animals;
2

select sum (distinct weight) from animals;
4

Mixing Aggregate Functions and Single Values

An aggregate function also selects a row in the table

```
select max(weight), kind from animal;
        1200 | t-rex
select min(kind), kind from animals;
        cat|cat|cat
select max(legs), kind from animals;
        4| cat      * no clear answer *
select avg(weight), kind from animals;
        2009.3 | t-rex
```

Groups

Grouping Rows

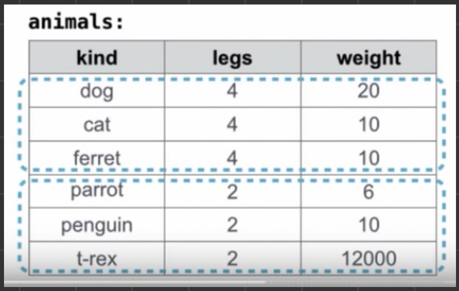
Rows in a table can be grouped, and aggregation is performed on each group

```
select [columns] from [table] group by [expression]
    having [expression];
```

The number of groups is the number of unique values of an expression

```
select legs, max(weight) from animals group by legs;
```

legs	max(weight)
4	20
2	12 000

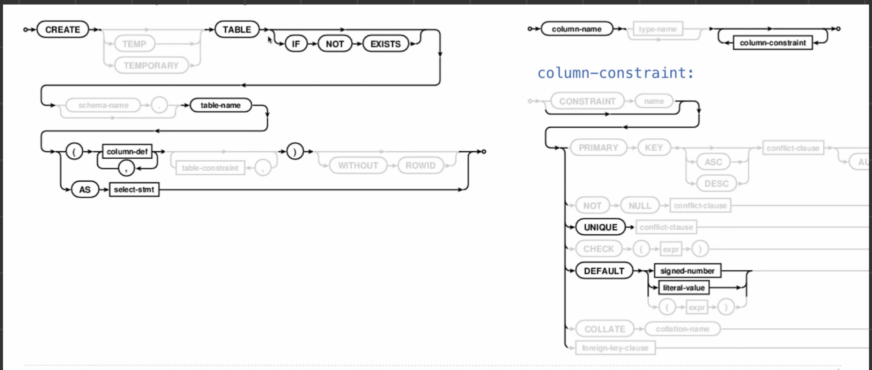


animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

11/22/19 - Databases

Create Table



Examples:

CREATE TABLE numbers (n, note):

↑ every n gets a note

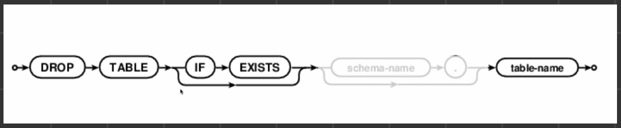
CREATE TABLE numbers (n UNIQUE, note):

↑ every n gets unique note only

CREATE TABLE numbers (n, note DEFAULT "no comment")

↑ default comment is none

DROP table



Insert

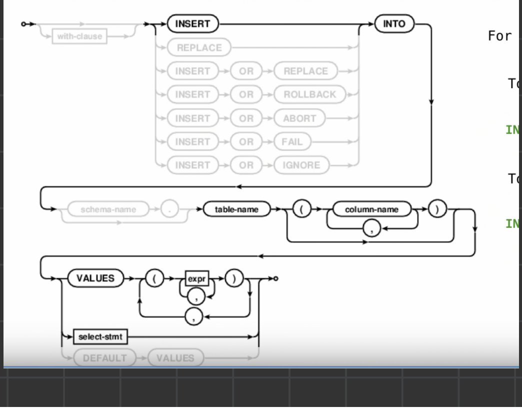
For a table t with 2 columns

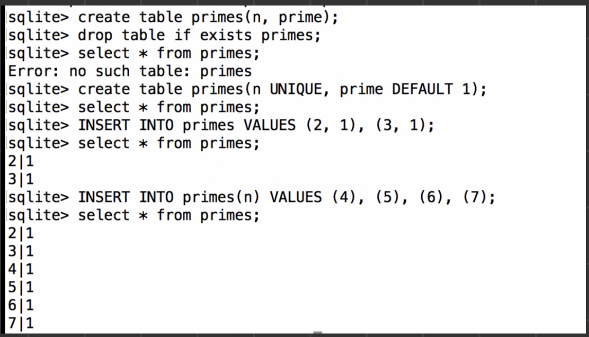
to insert into 1 column:

insert into t (column) values (value):

to insert into both columns:

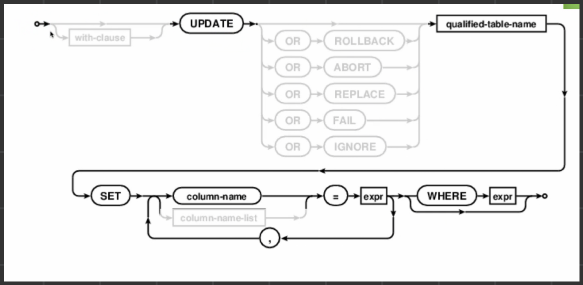
insert into t values (value0, value1):





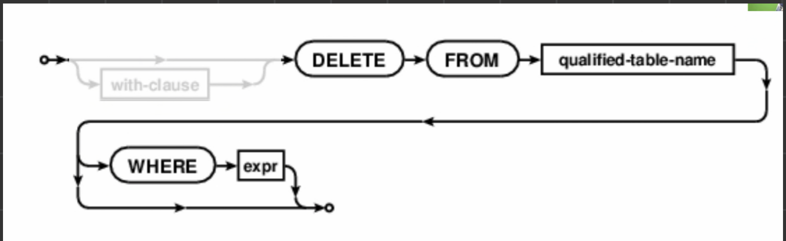
```
sqlite> create table primes(n, prime);
sqlite> drop table if exists primes;
sqlite> select * from primes;
Error: no such table: primes
sqlite> create table primes(n UNIQUE, prime DEFAULT 1);
sqlite> select * from primes;
sqlite> INSERT INTO primes VALUES (2, 1), (3, 1);
sqlite> select * from primes;
2|1
3|1
sqlite> INSERT INTO primes(n) VALUES (4), (5), (6), (7);
sqlite> select * from primes;
2|1
3|1
4|1
5|1
6|1
7|1
```

Update



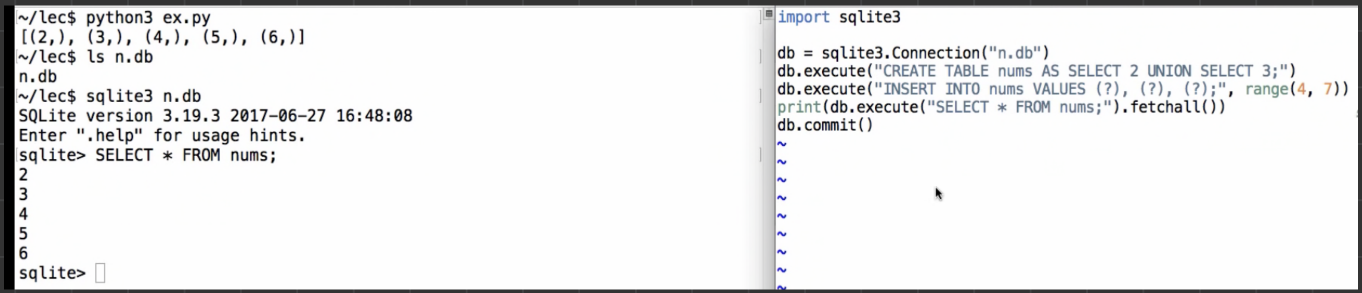
update primes SET prime=0 where n>2 and n%2=0

Delete



delete from primes where prime=0;

Python & SQL



```
~/lec$ python3 ex.py
[(2,), (3,), (4,), (5,), (6,)]
~/lec$ ls n.db
n.db
~/lec$ sqlite3 n.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> SELECT * FROM nums;
2
3
4
5
6
sqlite> 
```

```
import sqlite3

db = sqlite3.Connection("n.db")
db.execute("CREATE TABLE nums AS SELECT 2 UNION SELECT 3;")
db.execute("INSERT INTO nums VALUES (?), (?), (?);", range(4, 7))
print(db.execute("SELECT * FROM nums;").fetchall())
db.commit()
~
~
~
~
~
~
~
~
```

SQL injection attack

name = "Robert'); Drop table students; --"

cmd = 'INSERT INTO students VALUES ('" + name + "');"

db.executescript(cmd)

↳ insert into students VALUES ('Robert'); Drop table students; --');

would become deleted

* instead * db.execute("insert into students values (?)", [name])