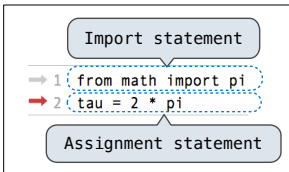
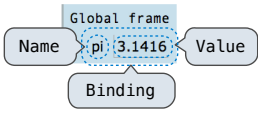




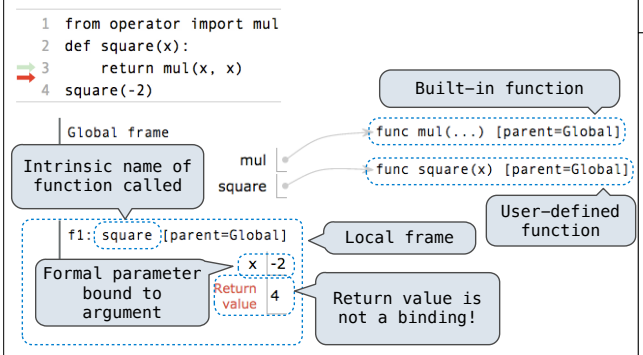
Code (left):

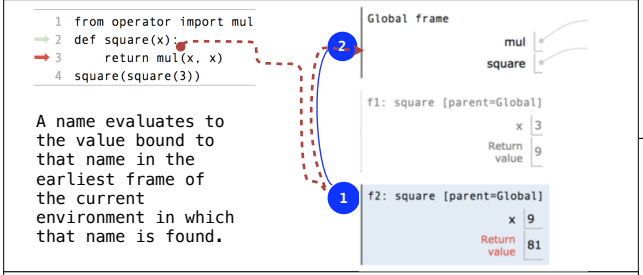
Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

A name is bound to a value

In a frame, there is at most one binding per name





Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:

1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Execution rule for def statements:

1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:

1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for and expressions:

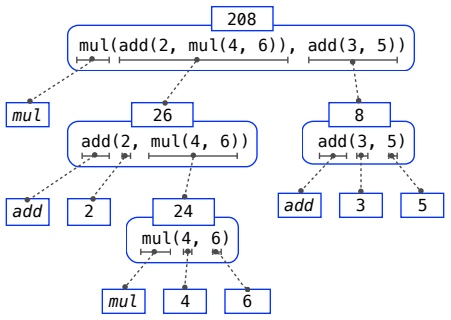
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.



Pure Functions

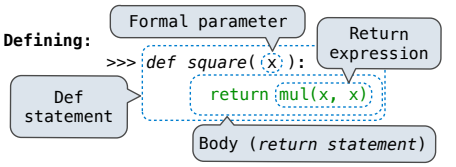
-2 ▶ *abs*(number): ▶ 2

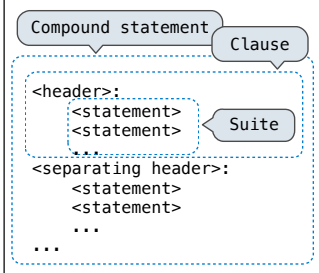
2, 10 ▶ *pow*(x, y): ▶ 1024

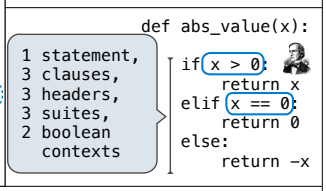
Non-Pure Functions

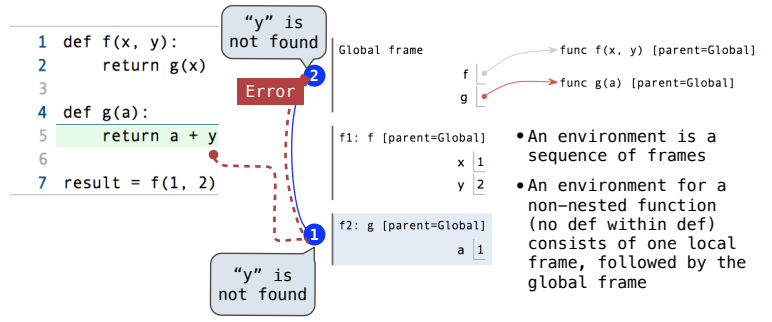
-2 ▶ *print*(...): ▶ None

display "-2"

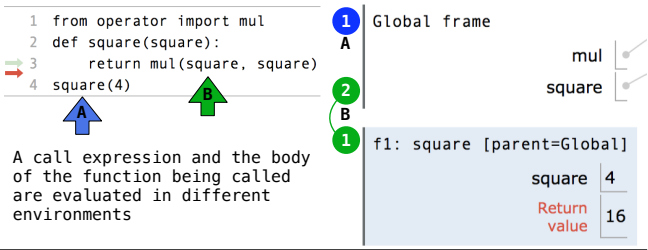


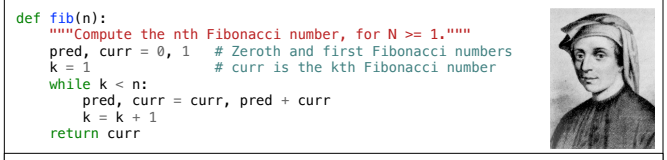




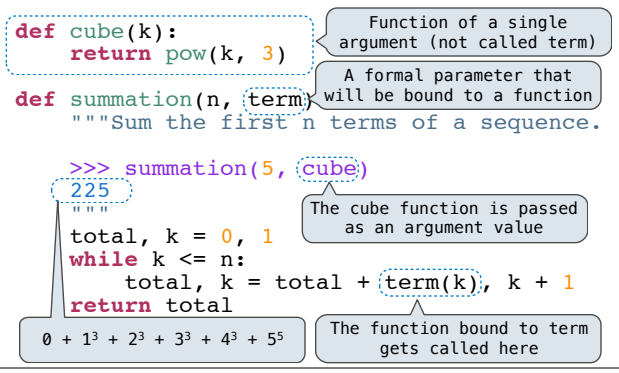


- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame





```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1   # Zeroth and first Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Nested def statements: Functions defined within other function bodies are bound to names in the local frame

```
square = lambda x,y: x * y
```

A function with formal parameters x and y that returns the value of "x * y"

Evaluates to a function. No "return" keyword!

Must be a single expression

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

A function that returns a function

The name add_three is bound to a function

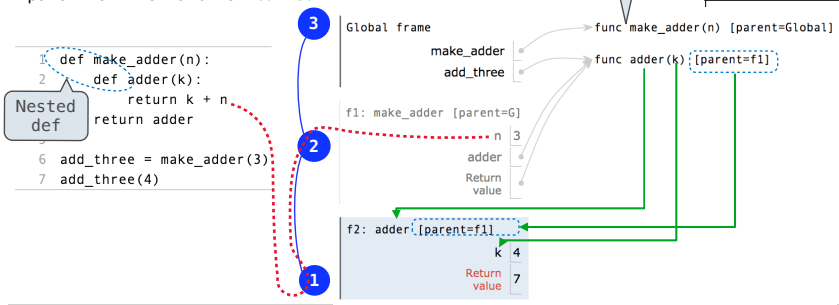
A local def statement

Can refer to names in the enclosing function

- Every user-defined **function** has a *parent frame* (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a *parent frame* (often global)
- The parent of a **frame** is the parent of the function *called*

A function's signature has all the information to create a local frame

Nested def




```
def curry2(f):
    """Returns a function g such that g(x)(y) returns f(x, y)."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Anatomy of a recursive function:

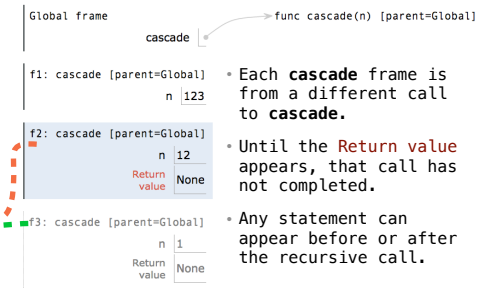
- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
cascade(123)
```

Program output:

```
123
12
1
12
```



- Each **cascade** frame is from a different call to **cascade**.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow = lambda n: f_then_g(grow, print, n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

```
n:      0, 1, 2, 3, 4, 5, 6,  7,  8,
fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21,

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

`square = lambda x: x * x` **VS**

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

When a function is defined:

1. Create a **function value**: func <name>(<formal parameters>)
2. Its parent is the current frame.

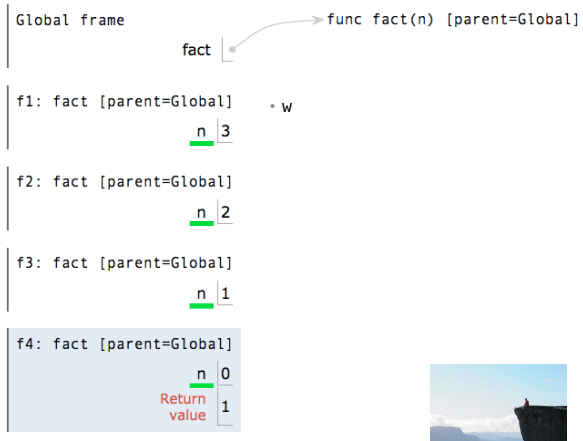
f1: make_adder func adder(k) [parent=f1]

3. Bind <name> to the **function value** in the current frame (which is the first frame of the current environment).

When a function is called:

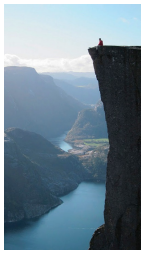
1. Add a **local frame**, titled with the <name> of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts with the **local frame**.

```
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```



Is fact implemented correctly?

1. Verify the base case.
2. Treat fact as a functional abstraction!
3. Assume that fact(n-1) is correct.
4. Verify that fact(n) is correct, assuming that fact(n-1) correct.



- Recursive decomposition: finding simpler instances of a problem.
- E.g., **count_partitions(6, 4)**
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - **count_partitions(2, 4)**
  - **count_partitions(6, 3)**
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

```
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas

lumeric types in Python:

```
>>> type(2)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

Represents integers exactly

Represents real numbers approximately

ational implementation using functions:

```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numer(x):
    return x('n')

def denom(x):
    return x('d')
```

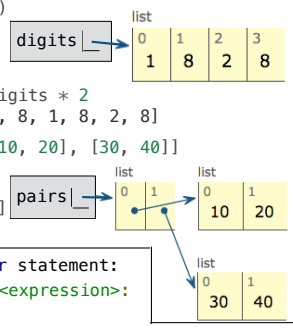
This function represents a rational number

Constructor is a higher-order function

Selector calls x

ists:

```
>> digits = [1, 8, 2, 8]
>> len(digits)
>> digits[3]
>> [2, 7] + digits * 2
, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>> pairs = [[10, 20], [30, 40]]
>> pairs[1]
0, 40]
>> pairs[1][0]
```



xecuting a for statement:

```
or <name> in <expression>:
    <suite>
```

. Evaluate the header <expression>, which must yield an iterable value (a sequence)
. For each element in that sequence, in order:
 A. Bind <name> to that element in the current frame
 B. Execute the <suite>

npacking in a or statement:

A sequence of fixed-length sequences

A name for each element in a fixed-length sequence

```
>> pairs=[[1, 2], [2, 2], [3, 2], [4, 4]]
>> same_count = 0
>> for x, y in pairs:
..     if x == y:
..         same_count = same_count + 1
>> same_count
```

```
..., -3, -2, -1, 0, 1, 2, 3, 4, ...
         range(-2, 2)
```

ength: ending value - starting value
lement selection: starting value + index

```
>> list(range(-2, 2))
-2, -1, 0, 1]
>> list(range(4))
0, 1, 2, 3]
```

List constructor

Range with a 0 starting value

mbership:

```
> digits = [1, 8, 2, 8]
> 2 in digits
ue
> 1828 not in digits
ue
```

Slicing:

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

List comprehensions:

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

The result of calling **repr** on a value is what Python prints in an interactive session

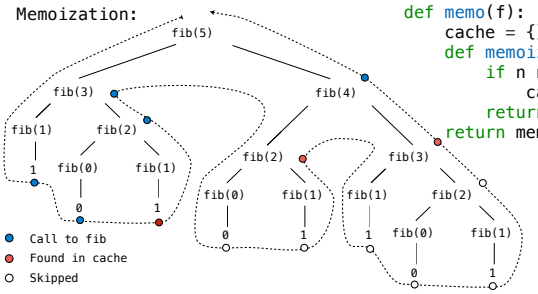
The result of calling **str** on a value is what Python prints using the **print** function

```
>>> 12e12
120000000000000.0
>>> print(repr(12e12))
120000000000000.0
>>> print(today)
2014-10-13
```

str and **repr** are both polymorphic; they apply to any object
repr invokes a zero-argument method **__repr__** on its argument

```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'
>>> today.__str__()
'2014-10-13'
```

Memoization:




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

Type dispatching: Look up a cross-type implementation of an operation based on the types of its arguments
Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

$R(n) = \Theta(f(n))$ means that there are positive constants k_1 and k_2 such that $k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$ for all n larger than some m

$\Theta(b^n)$ Exponential growth. Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

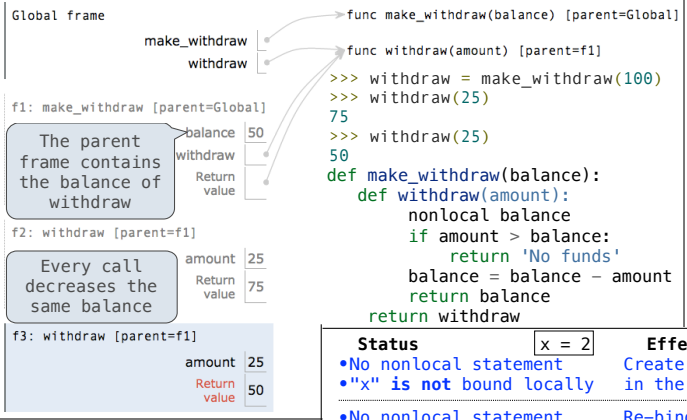
$\Theta(n^2)$ Quadratic growth. E.g., overlap
Incrementing n increases R(n) by the problem size n

$\Theta(n)$ Linear growth. E.g., factors or exp

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n)

$\Theta(1)$ Constant. The problem size doesn't matter



The parent frame contains the balance of withdraw

Every call decreases the same balance

```
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'No funds'
        balance = balance - amount
        return balance
    return withdraw
```

Strings as sequences:

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4]
False
```

List & dictionary mutation:

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

```
>>> nums = {'I': 1.0, 'V': 5, 'X': 10}
>>> nums['X']
10
>>> nums['I'] = 1
>>> nums['L'] = 50
>>> nums
{'X': 10, 'L': 50, 'V': 5, 'I': 1}
>>> sum(nums.values())
66
>>> dict([(3, 9), (4, 16), (5, 25)])
{3: 9, 4: 16, 5: 25}
>>> nums.get('A', 0)
0
>>> nums.get('V', 0)
5
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits
>>> suits.pop()
'myriad'
>>> suits.remove('string')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['heart', 'diamond']
>>> suits
['heart', 'diamond', 'spade', 'club']
>>> original_suits
['heart', 'diamond', 'spade', 'club']
```

Identity:
<exp0> **is** <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to the same object
Equality:
<exp0> == <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to equal values
Identical objects are always equal valu

You can copy a list by calling the list constructor or slicing the list from th beginning to the end.

Constants: Constant terms do not affect the order of growth of a process

$\Theta(n)$ $\Theta(500 \cdot n)$ $\Theta(\frac{1}{500} \cdot n)$

Logarithms: The base of a logarithm does not affect the order of growth of a proce

$\Theta(\log_2 n)$ $\Theta(\log_{10} n)$ $\Theta(\ln n)$

Nesting: When an inner process is repeate for each step in an outer process, multipl the steps in the outer and inner processe to find the total number of steps

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

Outer: length of

Inner: length of

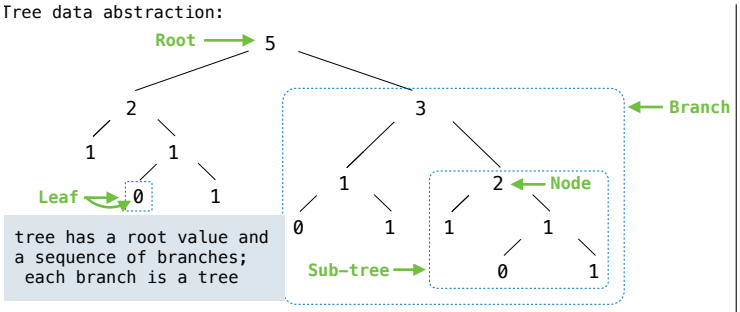
If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing pa of the computation dominates the total

$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 100$

Status	x = 2	Effect
•No nonlocal statement •"x" **is not** bound locally		Create a new binding from name "x" to number 2 in the first frame of the current environment
•No nonlocal statement •"x" **is** bound locally		Re-bind name "x" to object 2 in the first frame of the current environment
•nonlocal x •"x" **is** bound in a non-local frame		Re-bind "x" to 2 in the first non-local frame o the current environment in which "x" is bound
•nonlocal x •"x" **is not** bound in a non-local frame		SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •"x" **is** bound in a non-local frame •"x" also bound locally		SyntaxError: name 'x' is parameter and nonlocal

Tree data abstraction:



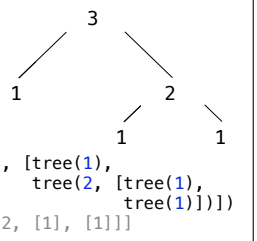
tree has a root value and a sequence of branches; each branch is a tree

```
def tree(root, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)
def root(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
def is_leaf(tree):
    return not branches(tree)
def leaves(tree):
    """The leaf values in tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list



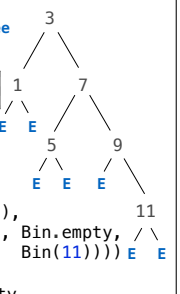
```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2),
        right = fib_tree(n-1)
        fib_n = root(left) + root(right)
        return tree(fib_n, [left, right])
```

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches

class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True
    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False
    @property
    def left(self):
        return self.branches[0]
    @property
    def right(self):
        return self.branches[1]
    def is_leaf(self):
        return self.left.is_empty and self.right.is_empty
```

Built-in isinstance function: returns True if branch has a class that *is* or *inherits from* Tree

E: An empty tree



```
Bin = BinaryTree
t = Bin(3, Bin(1),
           Bin(7, Bin(5),
                  Bin(9, Bin.empty,
                         Bin(11))))
```

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)

def extend_link(s, t):
    if s is Link.empty:
        return t
    else:
        return Link(s.first, extend_link(s.rest, t))

def map_link(f, s):
    if s is Link.empty:
        return s
    else:
        return Link(f(s.first), map_link(f, s.rest))
```

Some zero length sequence

Yes, this call is recursive

Sequence abstraction special names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

```
>>> s = Link(3, Link(4))
>>> extend_link(s, s)
Link(3, Link(4, Link(3, Link(4))))
>>> square = lambda x: x * x
>>> map_link(square, s)
Link(9, Link(16))
```

Python object system:

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

A new instance is created by calling a class

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

An account instance

balance: 0 holder: 'Jim'

When a class is called:
1. A new instance of that class is created:
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.

`__init__` is called a constructor

self should always be bound to an instance of the Account class or a subclass of Account

```
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 funds'
        self.balance = self.balance - amount
        return self.balance
```

Function call: all arguments within parentheses

Method invokation: One object before the dot and other arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Dot expression

Call expression

<expression> . <name>

The <expression> can be any valid Python expression.
The <name> must be a simple name.
Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

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 statements with a dot expression on their left-hand side affect attributes for the object of that dot expression
- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
        # or
        return super().withdraw(amount + self.withdraw_fee)
```

To look up a name in a class:
1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

CS 61A Final Exam Study Guide – Page 1

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. E.g.,
TypeError('Bad argument!')

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

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.

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

handling a <class 'ZeroDivisionError'>
>>> x
0
```

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the first frame of the current environment.
   B. Execute the <suite>.

An iterable object has a method __iter__ that returns an iterator.

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)
1
2
3
```

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
    except StopIteration:
        pass
```

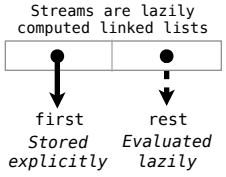
```
class FibIter:
    def __init__(self):
        self._next = 0
        self._addend = 1

    def __next__(self):
        result = self._next
        self._addend, self._next = self._next, self._addend + self._next
        return result
```

```
>>> fibs = FibIter()
>>> [next(fibs) for _ in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

"Please don't reference these directly. They may change."

A stream is a linked list, but the rest of the list is computed on demand.

Once created, Streams and Rlists can be used interchangeably using first and rest.




```
class Stream:
    """A lazily computed linked list."""
    class empty:
        def __repr__(self):
            return 'Stream.empty'
    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        assert callable(compute_rest), 'compute_rest must be callable.'
        self.first = first
        self._compute_rest = compute_rest

    @property
    def rest(self):
        """Return the rest of the stream, computing it if necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest

def integer_stream(first=1):
    def compute_rest():
        return integer_stream(first+1)
    return Stream(first, compute_rest)

def filter_stream(fn, s):
    if s is Stream.empty:
        return s
    def compute_rest():
        return filter_stream(fn, s.rest)
    if fn(s.first):
        return Stream(s.first, compute_rest)
    else:
        return compute_rest()

def primes(positives):
        def not_divisible(x):
            return x % positives.first != 0
        def compute_rest():
            return primes(filter_stream(not_divisible, positives.rest))
        return Stream(positives.first, compute_rest)
```

```
def map_stream(fn, s):
    if s is Stream.empty:
        return s
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first),
                  compute_rest)
```

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*. (lambda ...)

Dynamic scope: The parent of a frame is the environment in which a procedure was *called*. (mu ...)

```
> (define f (mu (x) (+ x y)))
> (define g (lambda (x y) (f (+ x x))))
> (g 3 7)
13
```

```
class LetterIter:
    def __init__(self, start='a', end='e'):
        self.next_letter = start
        self.end = end

    def __next__(self):
        if self.next_letter >= self.end:
            raise StopIteration
        result = self.next_letter
        self.next_letter = chr(ord(result)+1)
        return result

class Letters:
    def __init__(self, start='a', end='e'):
        self.start = start
        self.end = end

    def __iter__(self):
        return LetterIter(self.start, self.end)

def letters_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
        next_letter = chr(ord(next_letter)+1)
```

- A generator is an iterator backed by a generator function.
- Each time a generator function is called, it returns a generator.

```
>>> a_to_c = LetterIter('a', 'c')
>>> next(a_to_c)
'a'
>>> next(a_to_c)
'b'
>>> next(a_to_c)
Traceback (most recent call last):
    ...
StopIteration

>>> b_to_k = Letters('b', 'k')
>>> first_iterator =
b_to_k.__iter__()
>>> next(first_iterator)
'b'
>>> next(first_iterator)
'c'
>>> second_iterator = iter(b_to_k)
>>> second_iterator.__next__()
'b'
>>> first_iterator.__next__()
'd'

>>> for letter in
letters_generator('a', 'e'):
...     print(letter)
a
b
c
d
```

A **table** has columns and rows

A **column** has a name and a type

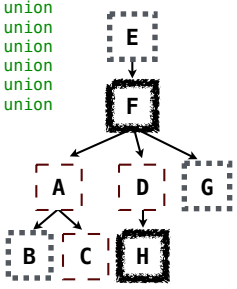
Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A **row** has a value for each column

```
select [expression] as [name], [expression] as [name], ...;
select [columns] from [table] where [condition] order by [order];
```

```
create table parents as
  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";
```

```
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";
```




```
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
barack	clinton
abraham	delano
abraham	grover
delano	grover

```
with
  ancestors(ancestor, descendent) as (
    select parent, child from parents union
    select ancestor, child
           from ancestors, parents
           where parent = descendent
  )
select ancestor from ancestors where descendent="herbert";
```

ancestor
delano
fillmore
eisenhower

```
create table pythagorean_triples as
  with
    i(n) as (
      select 1 union select n+1 from i where n < 20
    )
  select a.n as a, b.n as b, c.n as c
         from i as a, i as b, i as c
         where a.n < b.n and a.n*a.n + b.n*b.n = c.n*c.n;
```

a	b	c
3	4	5
5	12	13
6	8	10
8	15	17
9	12	15
12	16	20

The number of groups is the number of unique values of an expression
A having clause filters the set of groups that are aggregated

```
select weight/legs, count(*) from animals
                    group by weight/legs
                    having count(*)>1;
```

weight/legs	count(*)
5	2
2	2

weight/legs=5
weight/legs=2
weight/legs=2
weight/legs=3
weight/legs=5
weight/legs=6000

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

Scheme programs consist of expressions, which can be:
- Primitive expressions: 2, 3.3, true, +, quotient, ...
- Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; *symbols* are bound to values.
Call expressions have an operator and 0 or more operands.

A combination that is not a call expression is a *special form*:
- **If** expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

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

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

Lambda expressions evaluate to anonymous procedures.

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

λ

Two equivalent expressions:

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

An operator can be a combination too:

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

In the late 1950s, computer scientists used confusing names.
- **cons:** Two-argument procedure that **creates a pair**
- **car:** Procedure that returns the **first element** of a pair
- **cdr:** Procedure that returns the **second element** of a pair
- **nil:** The empty list

They also used a non-obvious notation for linked lists.
- A (linked) Scheme list is a pair in which the second element is nil or a Scheme list.
- Scheme lists are written as space-separated combinations.
- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

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

Not a well-formed list!

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists.

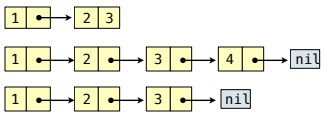
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```




```
class Pair:
    """A Pair has first and second attributes.

    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

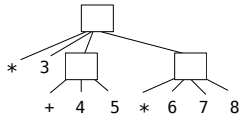
```
>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
```

The Calculator language has primitive expressions and call expressions

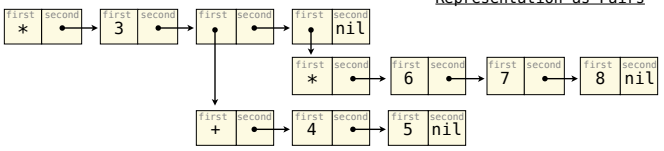
Calculator Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

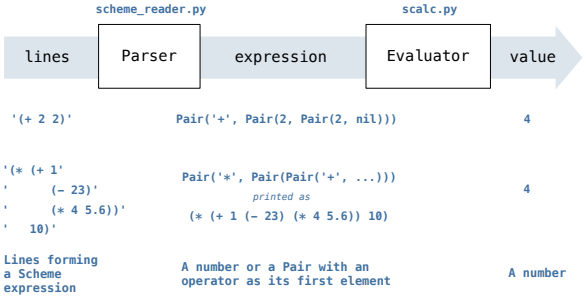
Expression Tree



Representation as Pairs



A basic interpreter has two parts: a *parser* and an *evaluator*.



A Scheme list is written as elements in parentheses:

(<element$_0$> <element$_1$> ... <element$_n$>)

A Scheme list

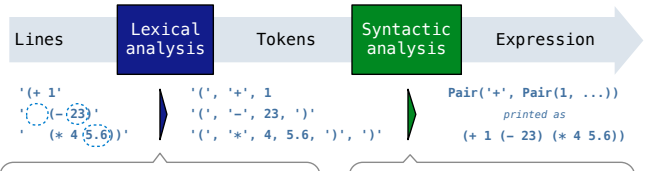
Each <element> can be a combination or atom (primitive).

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

A Parser takes a sequence of lines and returns an expression.



- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

Base case: symbols and numbers
Recursive call: scheme_read sub-expressions and combine them

The structure of the Scheme interpreter

Eval

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Creates a new environment each time a user-defined procedure is applied

Requires an environment for name lookup

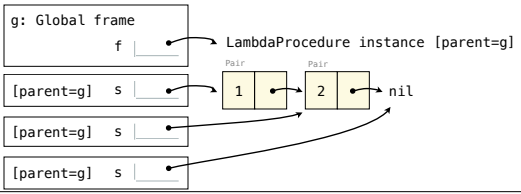
Apply

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure, then evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```



A procedure call that has not yet returned is *active*. Some procedure calls are *tail calls*. A Scheme interpreter should support an unbounded number of active tail calls.
A tail call is a call expression in a *tail context*, which are:
- The last body expression in a **lambda** expression
- Expressions 2 & 3 (consequent & alternative) in a tail context **if** expression

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

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

Not a tail call

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))) )
  (length-iter s 0) )
```

Recursive call is a tail call

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper