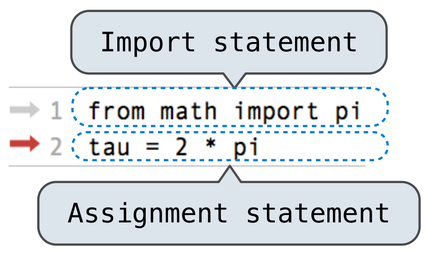
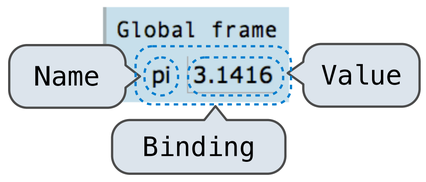




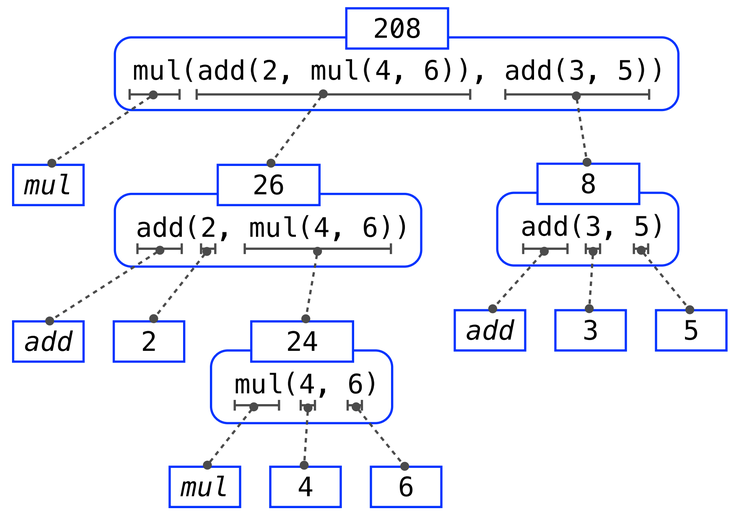
Code (left):

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

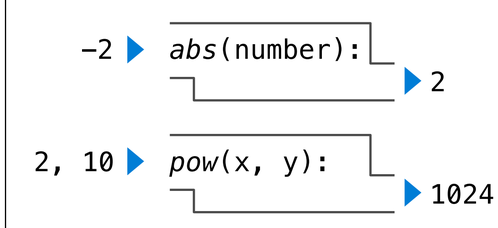
Frames (right):

A name is bound to a value

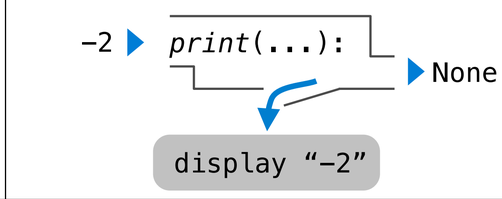
In a frame, there is at most one binding per name



Pure Functions



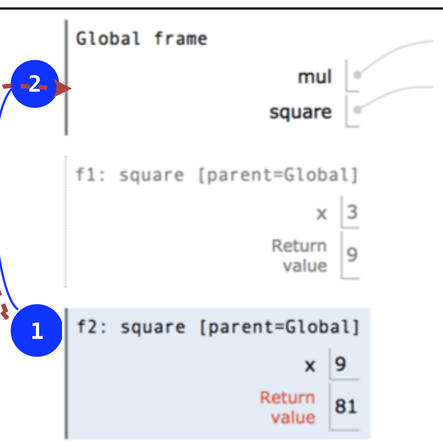
Non-Pure Functions




```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```


```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.



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>.

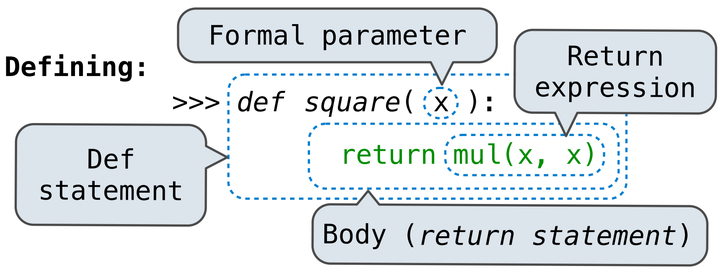
Evaluation rule for not expressions:

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

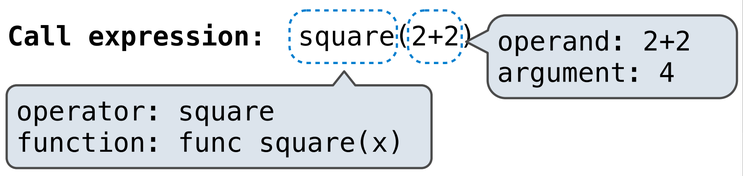
Execution rule for while statements:

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

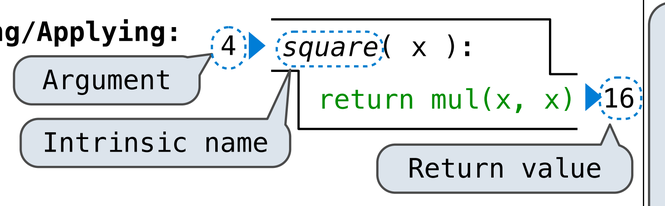
Defining:



Call expression:



Calling/Applying:




```
def strconcat( a, b ):
    print( a + " " + b )

strconcat( "hello", "world" )
```

```
hello world
```

```
A and B:
  True if A is True and B is True
A or B:
  True if A is True or B is True
not A:
  True if A is False
  False if A is True
```

```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

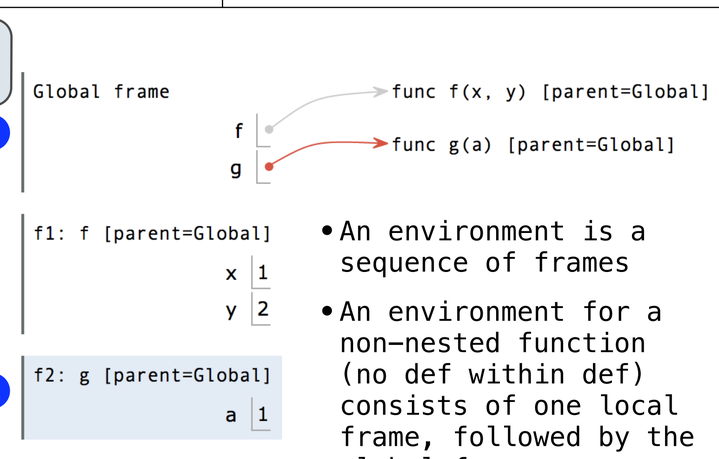
1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

```
1 def f(x, y):
2     return g(x)
3
4 def g(a):
5     return a + y
6
7 result = f(1, 2)
```

“y” is not found

Error

“y” is not found



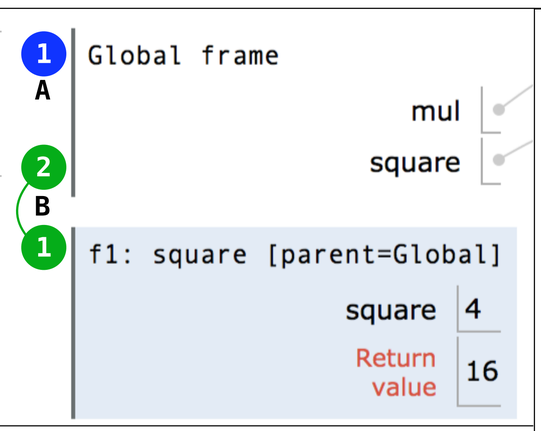
- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

```
1 from operator import mul
2 def square(square):
3     return mul(square, square)
4 square(4)
```

A

B

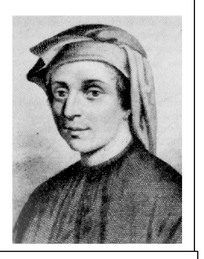
A call expression and the body of the function being called are evaluated in different environments



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

```
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
```



```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$

The function bound to term gets called here

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

Evaluates to a function. No "return" keyword!

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

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

```
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:

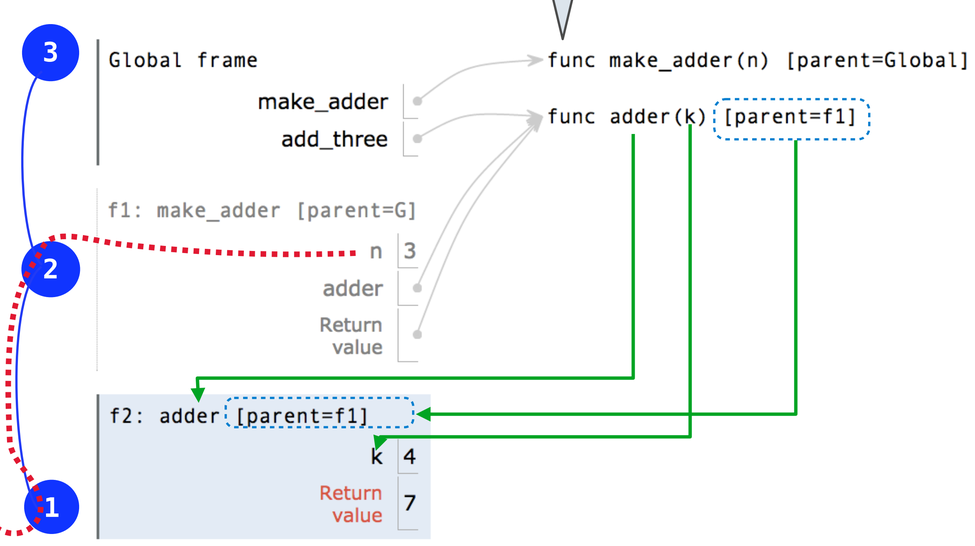
1. Add a **local frame**, titled with the <*name*> of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=<*label*>]
3. Bind the <*formal parameters*> to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts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

- 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

```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder
5
6 add_three = make_adder(3)
7 add_three(4)
```

Nested def

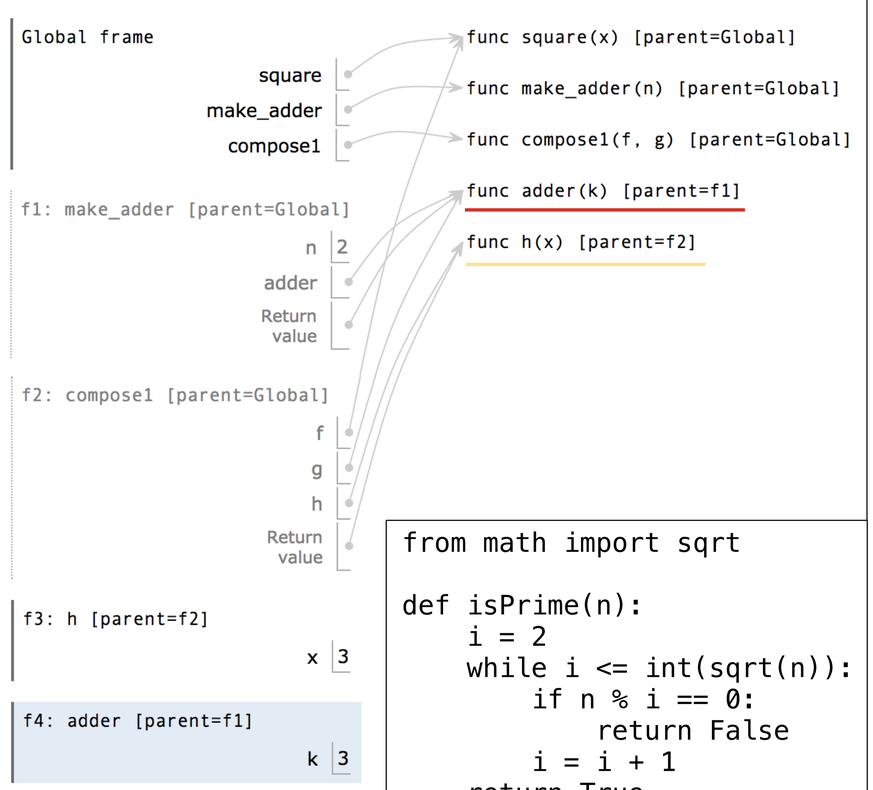



```
>>> min(2, 1, 4, 3)
1
>>> max(2, 1, 4, 3)
4
>>> abs(-2)
2
>>> pow(2, 3)
8
>>> len('word')
4
>>> round(1.75)
2
>>> print(1, 2)
1 2
>>> float(5)
5.0
```

```
>>> 2 + 3
5
>>> 2 * 3
6
>>> 2 ** 3
8
>>> 5 / 3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> str(5)
'5'
>>> int('5')
5
```

```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

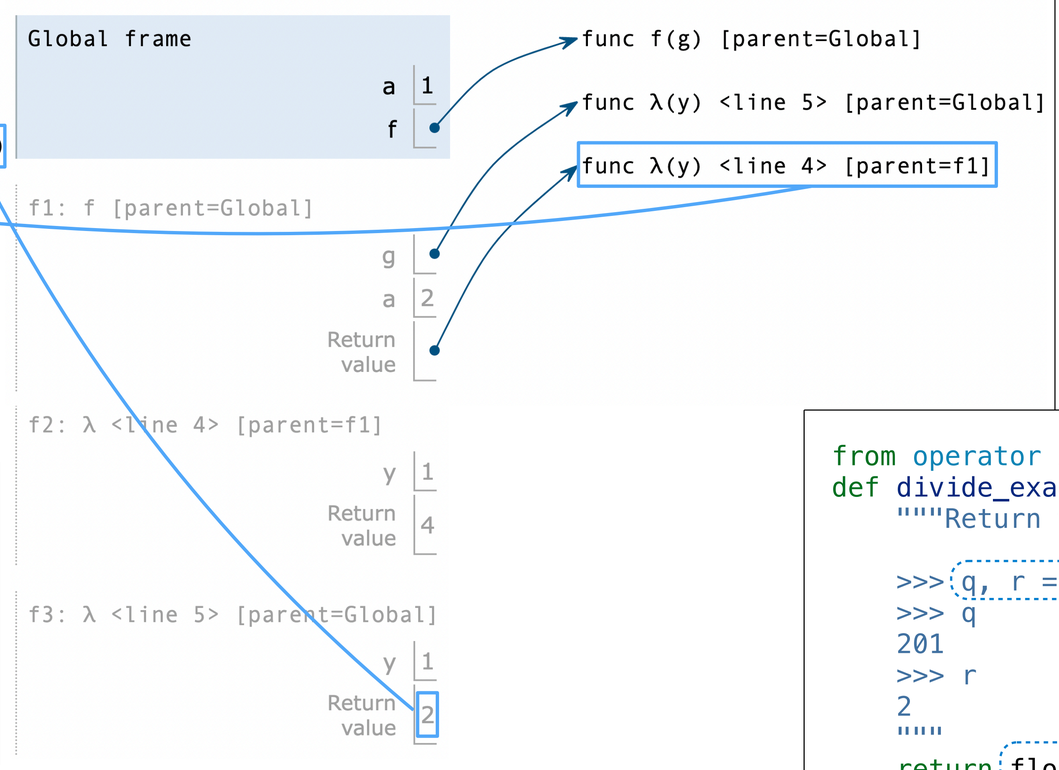



```
def search(f):
    """Return the smallest non-negative
    integer x for which f(x) is a true value.
    """
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def is_three(x):
    """Return whether x is three.

    >>> search(is_three)
    3
    """
    return x == 3

def inverse(f):
    """Return a function g(y) that returns
    x such that f(x) == y.

    >>> sqrt = inverse(lambda x: x * x)
    >>> sqrt(16)
    4
    """
    return lambda y: search(lambda x: f(x)==y)
```

```
from math import sqrt

def isPrime(n):
    i = 2
    while i <= int(sqrt(n)):
        if n % i == 0:
            return False
        i = i + 1
    return True
```

Frames Objects

```
1 a = 1
2 def f(g):
3     a = 2
4     return lambda y: a * g(y)
5 f(lambda y: a + y)(a)
```



A good coding practice:
1.) think, think, think
2.) sketch
3.) think more
4.) write 1-2 lines of code
5.) test your code
6.) test your code
7.) test your code
8.) goto step 4

False values so far: **0, False, '', None**

Anything value that's not false is true.

```
>>> if 0:
...     print('*')
>>> if 1:
...     print('*')
*
>>> if abs:
...     print('*')
*
```

```
>>> if 1 and 0:
...     print('*')
>>> if 1 or 0:
...     print('*')
*
>>> if 1 or 1/0:
...     print('*')
*
```

```
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

Two return values, separated by commas

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

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

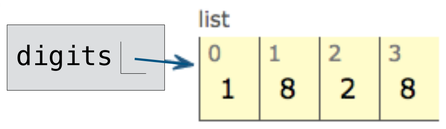
```
>>> today = datetime.date(2019, 10, 13)
>>> repr(today) # or today.__repr__()
'datetime.date(2019, 10, 13)'
>>> str(today) # or today.__str__()
'2019-10-13'
```

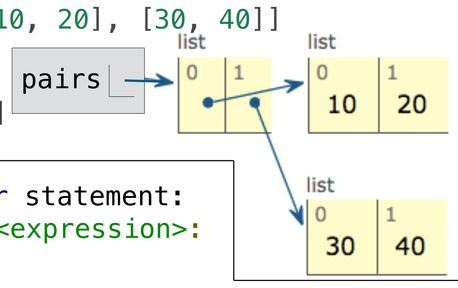
The result of evaluating an f-string literal contains the str string of the value of each sub-expression.

```
>>> f'pi starts with {pi}...'
'pi starts with 3.141592653589793...'
>>> print(f'pi starts with {pi}...')
pi starts with 3.141592653589793...
```

Lists:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```





Executing a `for` statement:

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

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

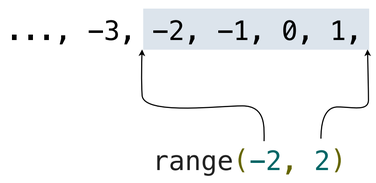
Unpacking in a for statement:

A sequence of fixed-length sequences

```
>>> pairs=[[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1
>>> same_count
2
```



Length: ending value − starting value

Element selection: starting value + index

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

List constructor

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

Range with a 0 starting value

Membership:

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

Slicing:

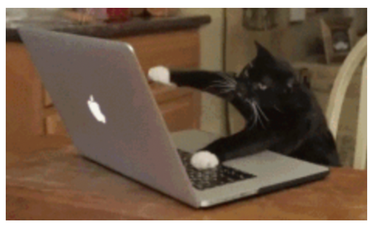
```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

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 values



```
iter(iterable):
  Return an iterator
  over the elements of
  an iterable value
next(iterator):
  Return the next element
```

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
```

*A generator function is a function that **yield**s values instead of **return**ing.*

```
>>> def plus_minus(x):
...     yield x
...     yield -x
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

```
def a_then_b(a, b):
    yield from a
    yield from b
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

List comprehensions:

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

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

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

Dictionaries:

```
words = {
        "más": "more",
        "otro": "other",
        "agua": "water"
}
```

```
>>> len(words)
3
>>> "agua" in words
True
>>> words["otro"]
'other'
>>> words["pavo"]
KeyError
>>> words.get("pavo", "🤔")
'🤔'
```

Dictionary comprehensions:

`{key: value for <name> in <iter exp>}`

```
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

```
>>> [word for word in words]
['más', 'otro', 'agua']
>>> [words[word] for word in words]
['more', 'other', 'water']
>>> words["oruguita"] = 'caterpillar'
>>> words["oruguita"]
'caterpillar'
>>> words["oruguita"] += '🐛'
>>> words["oruguita"]
'caterpillar🐛'
```

Functions that aggregate iterable arguments

- **sum**(iterable[, start]) -> value — *sum of all values*
- **max**(iterable[, key=func]) -> value — *largest value*
  max(a, b, c, ...[, key=func]) -> value
  min(iterable[, key=func]) -> value — *smallest value*
  min(a, b, c, ...[, key=func]) -> value
- **all**(iterable) -> bool — *whether all are true*
  any(iterable) -> bool — *whether any is true*

Many built-in Python sequence operations return iterators that compute results lazily

```
map(func, iterable):
    Iterate over func(x) for x in iterable
filter(func, iterable):
    Iterate over x in iterable if func(x)
zip(first_iter, second_iter):
    Iterate over co-indexed (x, y) pairs
reversed(sequence):
    Iterate over x in a sequence in reverse order
```

To view the contents of an iterator, place the resulting elements into a container

```
list(iterable):
    Create a list containing all x in iterable
tuple(iterable):
    Create a tuple containing all x in iterable
sorted(iterable):
    Create a sorted list containing x in iterable
```

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

```
>>> cascade(123)
123
12
1
12
123
```

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

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

Exponential growth. E.g., recursive fib — $\Theta(b^n)$ $O(b^n)$
Incrementing *n* multiplies *time* by a constant

Quadratic growth. E.g., overlap — $\Theta(n^2)$ $O(n^2)$
Incrementing *n* increases *time* by *n* times a constant

Linear growth. E.g., slow exp — $\Theta(n)$ $O(n)$
Incrementing *n* increases *time* by a constant

Logarithmic growth. E.g., exp_fast — $\Theta(\log n)$ $O(\log n)$
Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time — $\Theta(1)$ $O(1)$

List 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
```

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

```
>>> a = [10, 20, 30]
>>> list(a)
[10, 20, 30]
>>> a[:]
[10, 20, 30]
```

Tuples:

```
>>> empty = ()
>>> len(empty)
0
>>> conditions = ('rain', 'shine')
>>> conditions[0]
'rain'
>>> conditions[0] = 'fog'
Error
```

```
>>> all([False, True])
False
>>> all([])
True
>>> sum([1, 2])
3
>>> sum([1, 2], 3)
6
>>> sum([])
0
>>> sum([[1], [2]], [])
[1, 2]
```

```
>>> any([False, True])
True
>>> any([])
False
>>> max(1, 2)
2
>>> max([1, 2])
2
>>> max([1, -2], key=abs)
-2
```

List methods:

```
>>> suits = ['coin', 'string', 'myriad']
>>> suits.pop()
'myriad'
```

Remove and return the last element

```
>>> suits.remove('string')
```

Removes first matching value

```
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
```

Add all values

```
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['diamond']
>>> suits
['diamond', 'spade', 'club']
>>> suits.insert(0, 'heart')
>>> suits
['heart', 'diamond', 'spade', 'club']
```

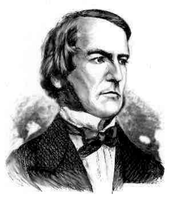
Replace a slice with values

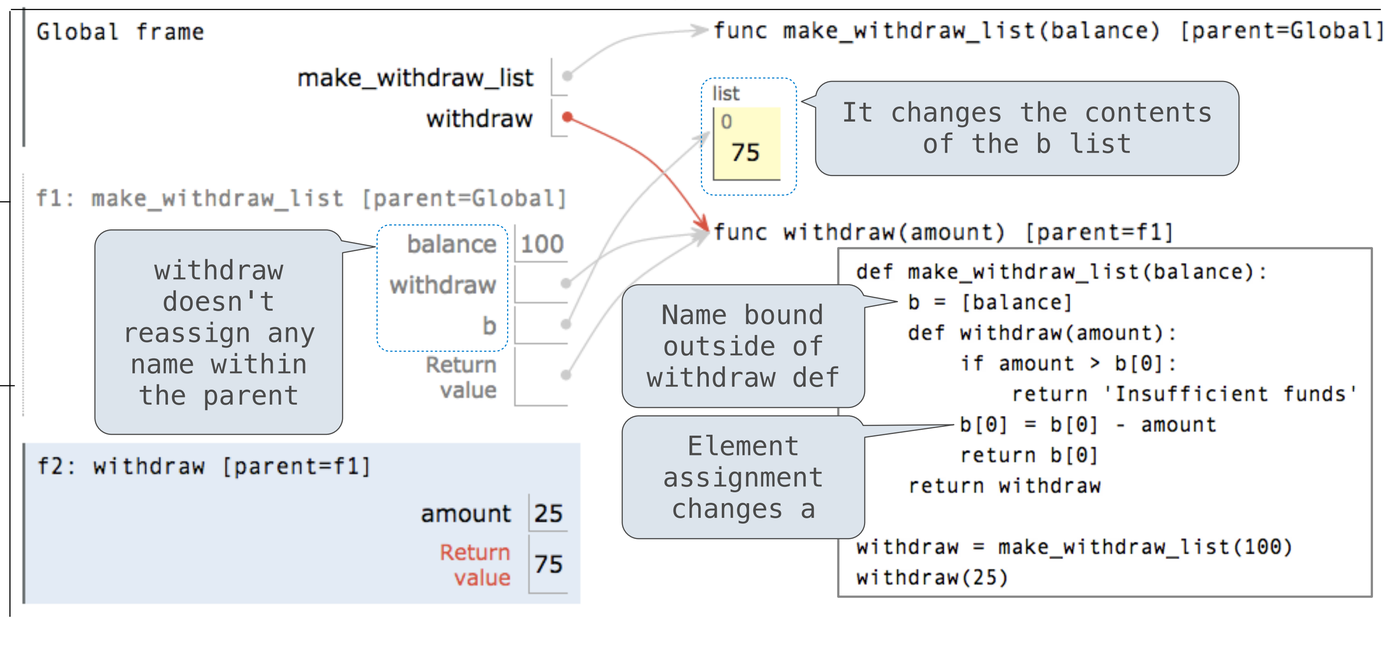
Add an element at an index

False values:
- Zero
- False
- None
- An empty string, list, dict, tuple

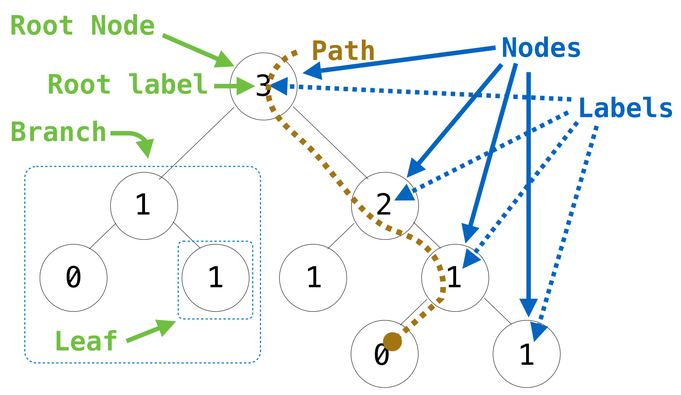
All other values are true values.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool('')
False
>>> bool('0')
True
>>> bool([])
False
>>> bool([[]])
True
>>> bool({})
False
>>> bool(())
False
>>> bool(lambda x: 0)
True
```





CS 61A Midterm Study Guide – Page 4



Recursive description:
- A **tree** has a root **label** and a list of **branches**
- Each branch is a **tree**
- A tree with zero branches is called a **leaf**

Relative description:
- Each location is a **node**
- Each **node** has a **label**
- One node can be the **parent/child** of another

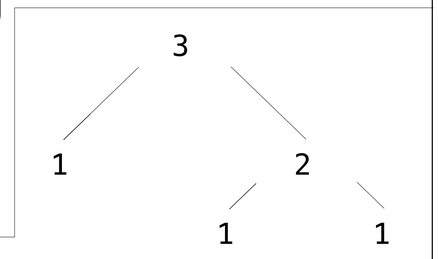
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
def is_leaf(tree):
    return not branches(tree)
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list

```
def leaves(t):
    """The leaf values in t.
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(t):
        return [label(t)]
    else:
        return sum([leaves(b) for b in branches(t)], [])
```




```
>>> 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 = label(left) + label(right)
        return tree(fib_n, [left, right])
```

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches
    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        def print_tree(t, indent=0):
            tree_str = '  ' * indent + str(t.label) + "\n"
            for b in t.branches:
                tree_str += print_tree(b, indent + 1)
            return tree_str
        return print_tree(self).rstrip()
```

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

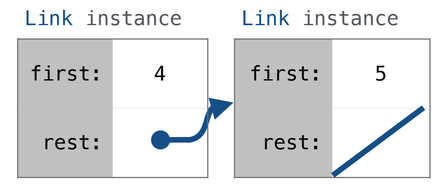
```
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_Tree(n-2)
        right = fib_Tree(n-1)
        fib_n = left.label+right.label
        return Tree(fib_n,[left, right])
```

```
def leaves(tree):
    "The leaf values in a tree."
    if tree.is_leaf():
        return [tree.label]
    else:
        return sum([leaves(b) for b in tree.branches], [])
```

CS 61A Final Exam Study Guide – Page 0

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest = ', ' + repr(self.rest)
        else:
            rest = ''
        return 'Link('+repr(self.first)+rest+')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Some zero length sequence




```
>>> s = Link(4, Link(5))
>>> s
Link(4, Link(5))
>>> s.first
4
>>> s.rest
Link(5)
>>> print(s)
<4 5>
>>> print(s.rest)
<5>
>>> s.rest.rest is Link.empty
True
```

Anatomy of a recursive function:
- The **def statement header** is like any function
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
    "Sum 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
```

- **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
```

CS 61A Final Exam Study Guide – Page 1

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 invocation: 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
```

Call expression

Dot 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
```

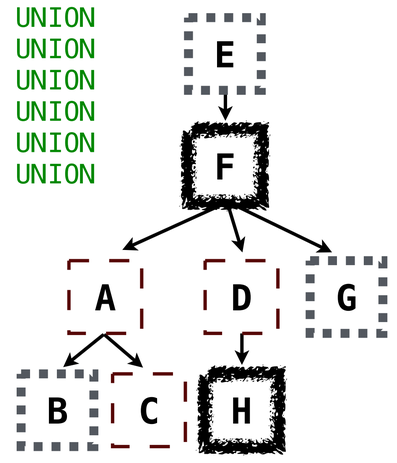
A **table** has columns and rows

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

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

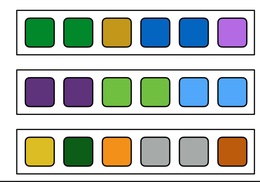
A **row** has a value for each column

```
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

```
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;
```



String values can be combined to form longer strings

```
sqlite> SELECT "hello," || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python

```
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
sqlite> SELECT substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1)
        FROM phrase;
low
```

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

An aggregate function in the [columns] clause computes a value from a group of rows:
- MAX([expression]) evaluates to the largest value of [expression] for any row in a group
- COUNT(*) evaluates to the number of rows in a group
- MIN, SUM, & AVG are also aggregate functions similar to MAX

With no GROUP BY clause, aggregation is performed over all rows:

```
select max(legs) from animals;
```

max(legs)
4

Scheme

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.

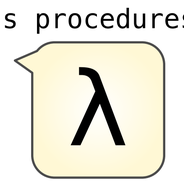
A combination that is not a call expression is a *special form*:
- **If** expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (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>)



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)
```

Scheme Lists

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

```
(car (cons 1 nil)) -> 1
(cdr (cons 1 nil)) -> ()
(cdr (cons 1 (cons 2 nil))) -> (2)
```

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)
```

Scheme Special Forms

```
(define size 5) ; =>  size
(if (> size 0) size (- size)) ; =>  5
(cond ((> size 0) size) ((= size 0) 0) (else (- size))) ; =>  5
(and (> size 1) (< size 10)) ; =>  #t
(or (> size 1) (< size 3)) ; =>  #t
(let ((a size) (b (+ 1 2))) (* 2 a b)) ; =>  30
(begin (define x (+ size 1)) (* x 2)) ; =>  12
((lambda (x y) (+ x y size)) size (+ 1 2)) ; =>  13
(define (add-two x y) (+ x y)) ; =>  add-two
`(+ size (- ,size) ,(* 3 4)) ; =>  (+ size (- 5) 12)
```

Scheme Built-In Procedures

```
(+ 2 5 1) ; =>  8
(- 9) ; =>  -9
(- 9 3 2) ; =>  4
(* 2 5) ; =>  10
(/ 7 2) ; =>  3.5
(/ 16 2 2 2) ; =>  2
(abs -1) ; =>  1
(remainder 7 3) ; =>  1
```

```
(null? '(1 2)) ; =>  #f
(null? '()) ; =>  #t
(= 1 2) ; =>  #f
(>= 2 1) ; =>  #t
(even? 2) ; =>  #t
(equal? '(1 2) '(1 2)) ; => #t
(eq? '(1 2) '(1 2)) ; => #f
(not (> 1 2)) ; =>  #t
```

```
(append '(1 2) '(3 4)) ; =>  (1 2 3 4)
(length '(1 2 3 4)) ; =>  4
(map (lambda (x) (+ x size)) '(2 3 4)) ; =>  (7 8 9)
(filter odd? '(2 3 4)) ; =>  (3)
(reduce + '(1 2 3 4 5)) ; =>  15
(list 1 2 3 4) ; =>  (1 2 3 4)
(list (cons 1 nil) size 'size) ; =>  ((1) 5 size)
(list (or #f #t) (or) (or 1 2)) ; =>  (#t #f 1)
(list (and #f #t) (and) (and 1 2)) ; =>  (#f #t 2)
(eval '(* 5 (* 4 (* 3 (* 2 (* 1 1)))))) ; => 120
(apply + '(1 2 3)) ; => 6
```

Scheme Scopes

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
```

Scheme Tail Calls

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**
- All final sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

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

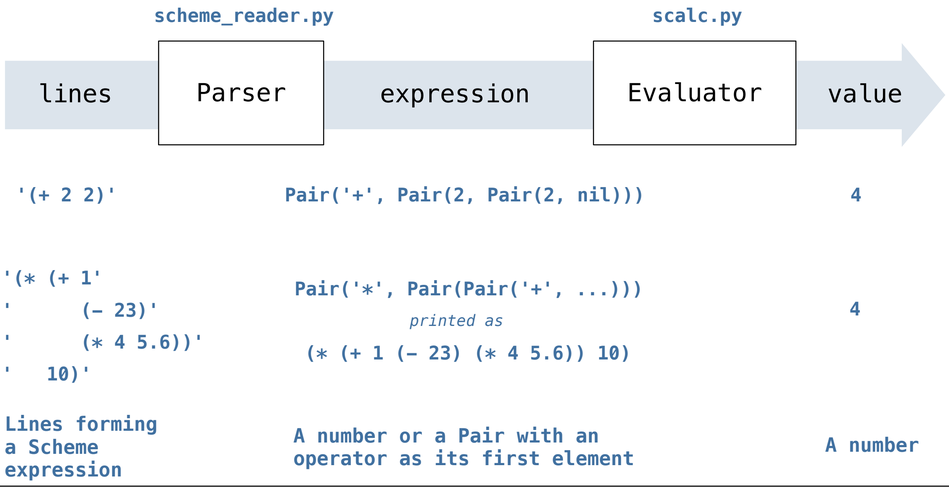
```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

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

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

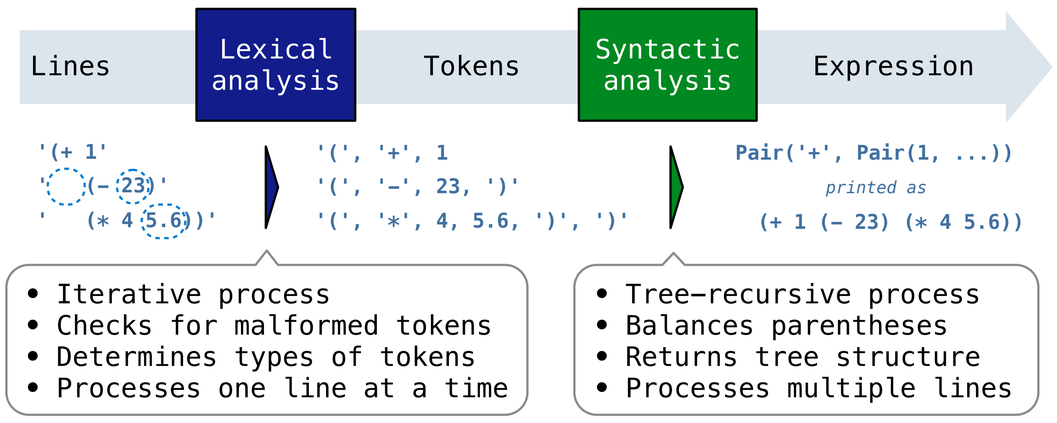
Each <element> can be a combination or atom (primitive).

(+ (* 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.



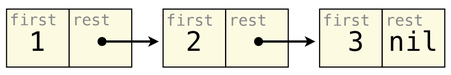
Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

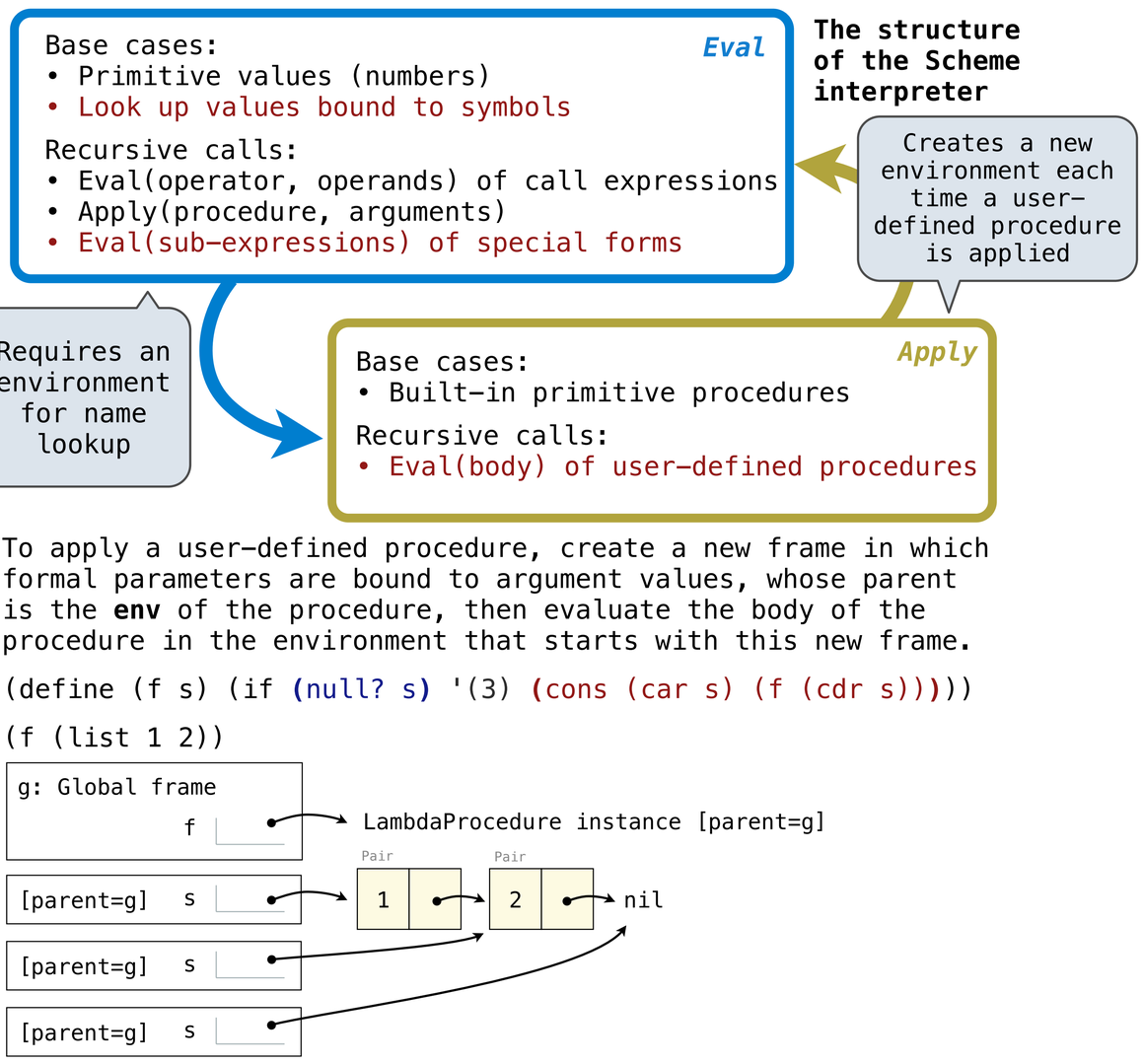
Each call to scheme_read consumes the input tokens for exactly one expression.

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

```
class Pair:
    """A pair has two instance attributes:
       first and rest.

       rest must be a Pair or nil.
    """
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest
```

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





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))
```

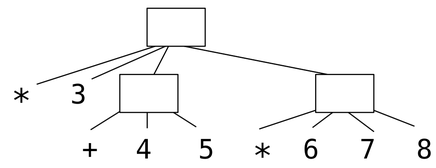

Calculator

The Calculator language has primitive expressions and call expressions

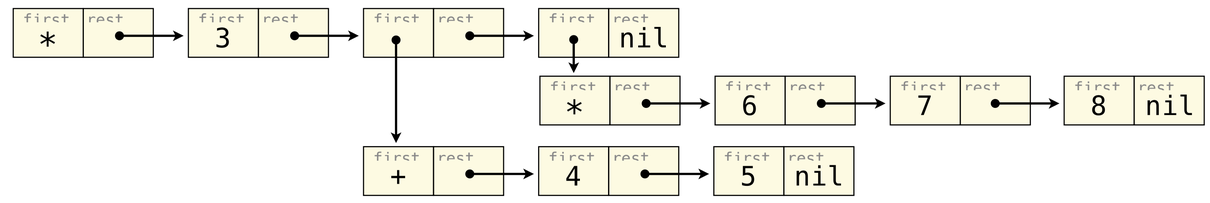
Calculator Expression

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

Expression Tree



Representation as Pairs



Scratch Paper

Feel free to use this space to do work. Work here will NOT be graded