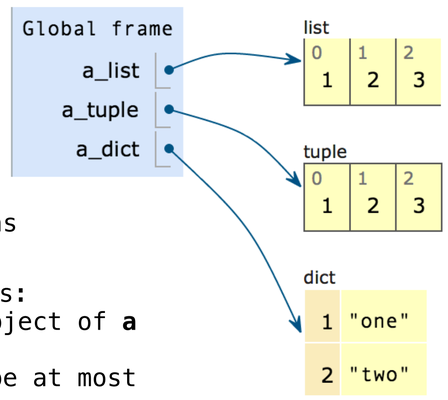
```
1  a_list = [1, 2, 3]
2  a_tuple = (1, 2, 3)
3  a_dict = {1: 'one', 2: 'two'}
```

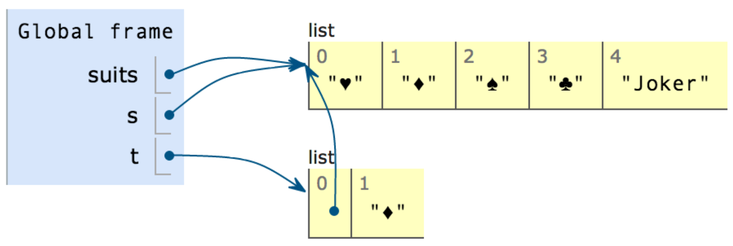


- Tuples are immutable sequences.
- Lists are mutable sequences.
- Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** type.
- Two **keys cannot be equal.** There can be at most one value for a key.

```
suits = ['♥', '♦']
s = suits
t = list(suits)
suits += ['♠', '♣']
t[0] = suits
suits.append('Joker')
```




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

1. Evaluate the header <expression>, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the local environment.
   B. Execute the <suite>.

A range is a sequence of consecutive integers.*

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

range(-2, 2)

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Generator expressions

(<map exp> for <name> in <iter exp> if <filter exp>)

- Evaluates to an iterable object.
- <iter exp> is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

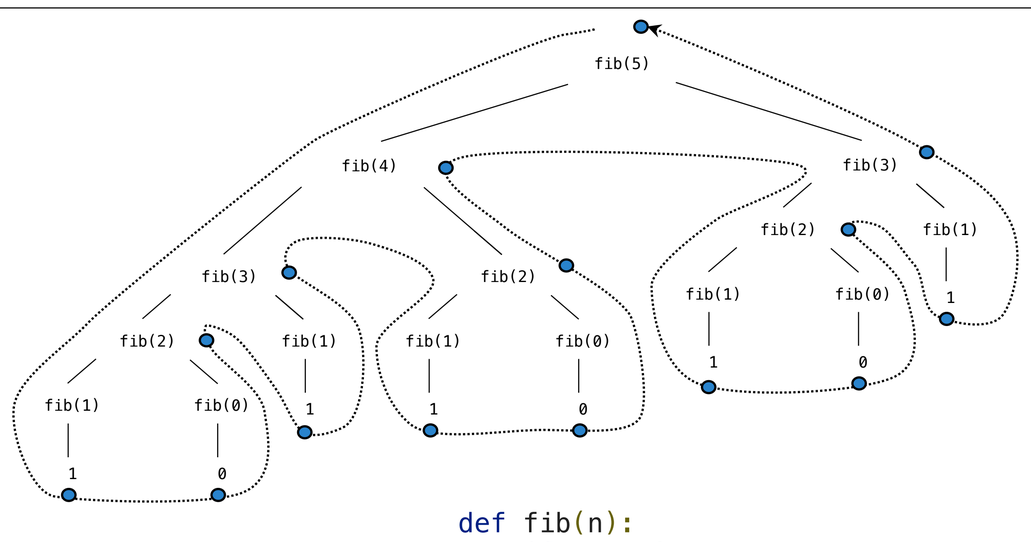
List comprehensions

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

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

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

```
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```




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

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

```
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment **does not** create a new object:

```
>>> a is a
True
>>> a is not b
True
>>> c = a
>>> c is a
True
```

```
nonlocal <name>, <name 2>, ...
```

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

From the Python 3 language reference:

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

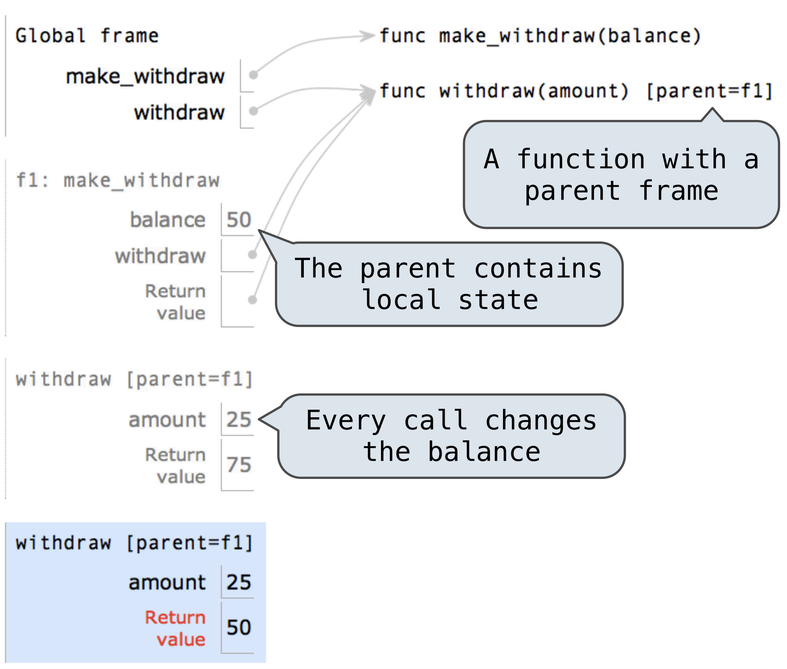
Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

`x = 2`

Status	Effect
• No nonlocal statement • "x" **is not** bound locally	Create a new binding from name "x" to object 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 env.
• nonlocal x • "x" **is** bound in a non-local frame (but not the global frame)	Re-bind "x" to 2 in the first non-local frame of the current environment in which it 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

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

withdraw = make_withdraw(100)
withdraw(25)
withdraw(25)
```



Python pre-computes which frame contains each name before executing the body of a function.

Therefore, within the body of a function, all instances of a name must refer to the same frame.

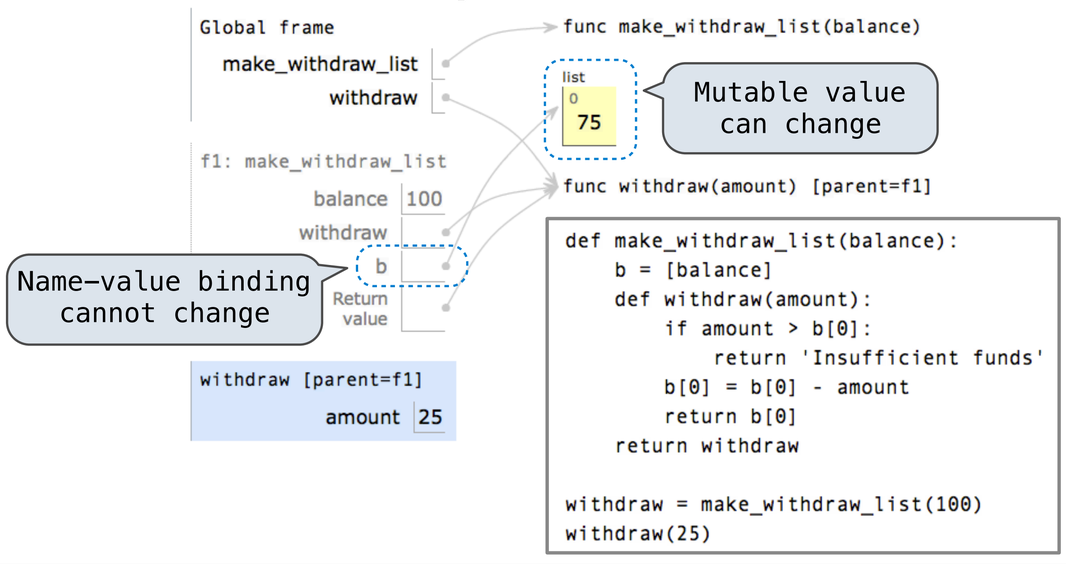
```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Mutable values can be changed *without* a nonlocal statement.




```
def pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])

def starts_with_a_vowel(w):
    return w[0].lower() in 'aeiou'
```

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

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

- A class statement **creates** a new class and **binds** that class to <name> in the first frame of the current environment.
- Statements in the <suite> create attributes of the class.

To evaluate a dot expression: <expression> . <name>

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.

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.

```
class Account(object):
    interest = 0.02
    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
```

Class attribute

Constructor

Methods

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

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

```
>>> jim_account.interest = 0.8
>>> jim_account.interest
0.8
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.8
```

Instance Attribute Assignment:

```
tom_account.interest = 0.08
```

This expression evaluates to an object

But the name ("interest") is **not** looked up

Attribute assignment statement adds or modifies the "interest" attribute of tom_account

---

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

Base class

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('T')
>>> ch.interest
0.01
>>> ch.deposit(20)
20
>>> ch.withdraw(5)
14
```

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1           # A free dollar!
```

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:
2. The constructor __init__ of the class is called with the new object as its first argument (called self), along with additional arguments provided in the call expression.

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

A mutable Rlist implementation using message passing

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

A bank account implemented using dispatch dictionaries

```
def account(balance):
    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']
    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']
    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}
    return dispatch
```

A simple container implemented using two accessor methods

```
def container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return put, get
```

```
class ComplexRI(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5
```

Special decorator: "Call this function on attribute look-up"

Type dispatching: Define a different function for each possible combination of types for which an operation is valid

```
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)

def isrational(z):
    return type(z) == Rational

def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numer/r.denom, z.imag)

def add_by_type_dispatching(z1, z2):
    """Add z1 and z2, which may be complex or rational."""
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)
```

Converted to a real number (float)

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
```

The base case

A recursive call

```
class Tree(object):
    def __init__(self, entry,
                 left=None,
                 right=None):
        self.entry = entry
        self.left = left
        self.right = right

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = map_rlist(s.rest, fn)
    return Rlist(fn(s.first),rest)

def count_entries(tree):
    if tree is None:
        return 0
    left = count_entries(tree.left)
    right = count_entries(tree.right)
    return 1 + left + right
```

A valid tree has no cycles!

Valid if these are None or a Tree instance

---

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, x = ty, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    key = (operator_name, tx)
    return coerce_apply.implementations[key](x, y)
```