

Rational implementation using functions:

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

This function represents a rational number

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

Constructor is a higher-order function

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

Selector calls x

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:

```
>>> 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
2
```

A name for each element in a fixed-length sequence

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

range(-2, 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]
```

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):

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

next(iterator):

Return the next element

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

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

```
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

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

```
>>> today = datetime.date(2019, 10, 13)
>>> print(today)
2019-10-13
```

```
>>> today.__repr__()
'datetime.date(2019, 10, 13)'
>>> today.__str__()
'2019-10-13'
```

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.

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 are 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(*iterables):
    Iterate over co-indexed (x, y) pairs
reversed(sequence):
    Iterate over x in a sequence in reverse order
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
```

To view the contents of an iterator,

place the resulting elements into a container

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

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

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

```
>>> fib(10)
0
1
1
2
3
5
8
13
21
```

Global frame

```
func make_withdraw(balance) [parent=Global]
make_withdraw
withdraw
balance 50
Return value 50
```

```
f1: make_withdraw [parent=Global]
The parent frame contains the balance of withdraw
```

```
f2: withdraw [parent=f1]
Every call decreases the same balance
amount 25
Return value 75
```

```
f3: withdraw [parent=f1]
amount 25
Return value 50
```

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}
```

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

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

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

```
>>> suits = ['coin', 'string', 'myriad']
>>> suits.pop()
'myriad'
>>> suits.remove('string')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> 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']
```

Remove and return the last element

Remove a value

Add all values

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({})
True
>>> bool(lambda x: 0)
True
```



Status

- No nonlocal statement
- "x" is not bound locally

- No nonlocal statement
- "x" is bound locally

- nonlocal x
- "x" is bound in a non-local frame

- nonlocal x
- "x" is not bound in a non-local frame

- nonlocal x
- "x" is bound in a non-local frame

- "x" also bound locally

Effect

Create a new binding from name "x" to number 2 in the first frame of the current environment

Re-bind name "x" to object 2 in the first frame of the current environment

Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" is bound

SyntaxError: no binding for nonlocal 'x' found

SyntaxError: name 'x' is parameter and nonlocal

Assignment Project Exam Help

https://powcoder.com

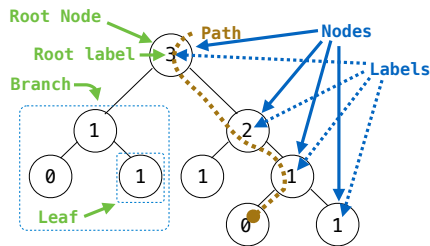
Add WeChat powcoder

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)
```

```
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)], [])
```

```
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 leaves(self):
```

```
        """The leaf values in a tree."
```

```
        if tree.is_leaf():
```

```
            return [tree.label]
```

```
        else:
```

```
            return sum([leaves(b) for b in tree.branches], [])
```

```
class Link:
```

```
    empty = ()
```

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

```
        assert rest is Link.empty or isinstance(rest, Link)
```

```
        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) + '>'
```

```
Link instance
```

```
Link instance
```

```
first: 4
```

```
rest: Link(5)
```

```
>>> 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

```
def count_partitions(n, m):
```

```
    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
```

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.

```
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
```

__init__ is called a constructor

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

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

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

```
>>> Account.deposit(a, 5)
```

```
10
```

```
>>> a.deposit(2)
```

```
12
```

Call expression

Dot expression

<expression> . <name>

The **<expression>** can be an arbitrary 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.
2. The object of the dot expression is 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: 0.00 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
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

