Rational 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')
```

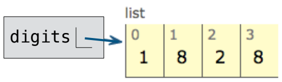
This function represents a rational number

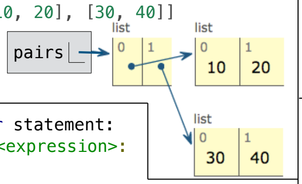
Constructor is a higher-order function

Selector calls x

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
```

..., -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]
```

Slicing creates a new object

Functions that aggregate iterable arguments

- **sum**(iterable[, start]) -> value
- **max**(iterable[, key=func]) -> value
  max(a, b, c, ...[, key=func]) -> value
  min(iterable[, key=func]) -> value
  min(a, b, c, ...[, key=func]) -> value
- **all**(iterable) -> bool
  any(iterable) -> bool

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
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

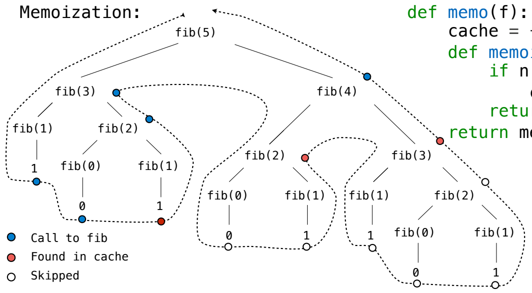
```
>>> print(today)
2014-10-13
```

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

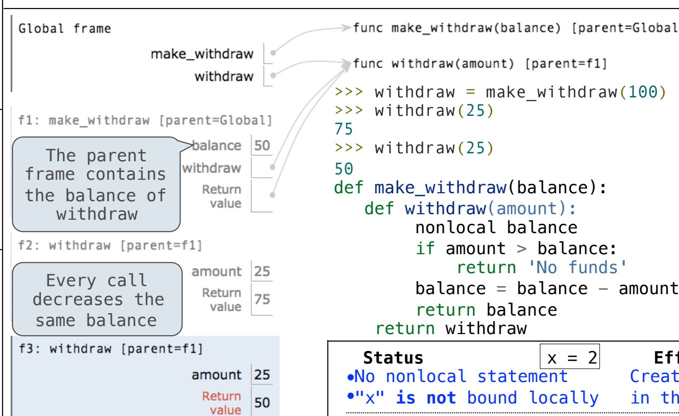
$\Theta(b^n)$ Exponential growth. Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$
Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ 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




```
>>> 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
```

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']
>>> 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

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

You can **copy** a list by calling the list constructor or slicing the list from the 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 process
$\Theta(\log_2 n)$ $\Theta(\log_{10} n)$ $\Theta(\ln n)$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes 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 a

Inner: length of b

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

Lower-order terms: The fastest-growing part of the computation dominates the total
$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

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 them.*

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

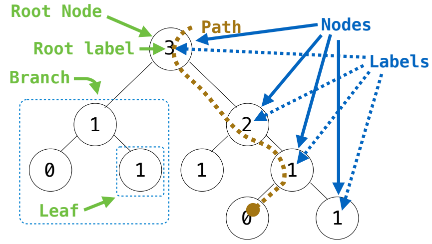
```
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

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

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 of 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

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

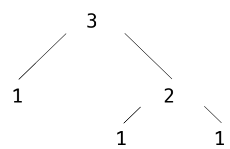



```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch 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)], [])
```

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 = 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 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], [])
```

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])
```

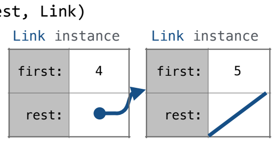
```
class Link:
    empty = ()

    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) + '>'
```

Some zero length sequence



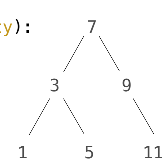

```
>>> s = Link(4, Link(5))
>>> s
Link(4, Link(5))
>>> s.first
4
>>> s.rest
Link(5)
>>> print(s)
<4, 5>
>>> print(s.rest)
<5>
>>> s.rest.rest is Link.empty
True
```

Python built-in sets:

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```

A binary search tree is a binary tree where each root is **larger** than all values in its left branch and **smaller** than all values in its right branch

```
class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])
    @property
    def left(self):
        return self.branches[0]
    @property
    def right(self):
        return self.branches[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
```

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

An account instance

balance: 0	holder: 'Jim'

2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

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

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

Function call: all arguments within parentheses

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 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
```