Numeric types in Python:

```
>>> type(2)
<class 'int'>
```
Represents integers exactly

```
>>> type(1.5)
<class 'float'>
```
Represents real numbers approximately

Data abstraction

Constructor

```
def rational(numer, denom):
    """Return a rational number."""
    return [numer, denom]
```

Selector

```
def numer(r):
    """Return the numerator of r."""
    return r[0]
```

Selector

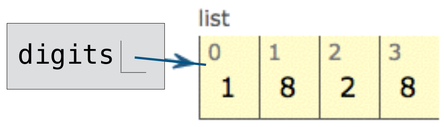
```
def denom(r):
    """Return the denominator of r."""
    return r[1]
```

Abstraction Barrier

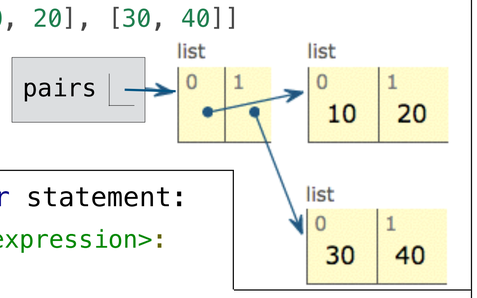
```
def add_rationals(r1, r2):
    num = numer(r1)*denom(r2) + \
          numer(r2)*denom(r1)
    den = denom(r1)*denom(r2)
    return rational(num, den)
```

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 sequence)
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

List comprehensions:

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

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

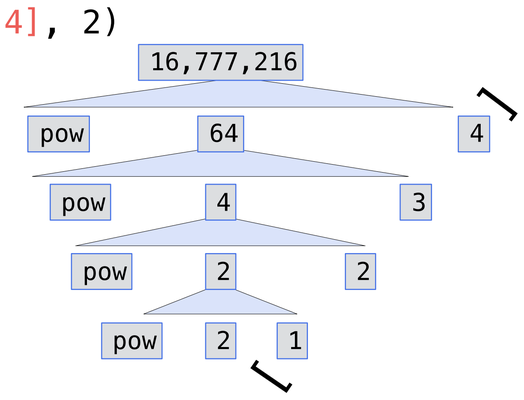
```
>>> x = map(lambda x: x * 3, [0, 1, 2, 3, 4])
>>> x
<map object at ..>
>>> list(x)
[0, 3, 6, 9, 12]
```

0, 1, 2, 3, 4 → λx: x*3 → 0, 3, 6, 9, 12

```
>>> y = filter(lambda x: x > 5, range(10))
>>> y
<filter object at ..>
>>> list(y)
[6, 7, 8, 9]
```

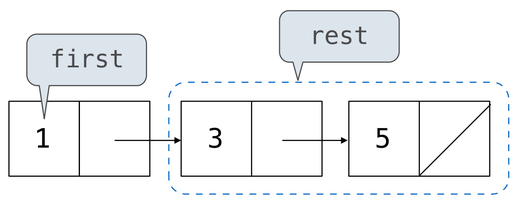
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 → λx: x>5 → 6, 7, 8, 9

```
>>> from functools import reduce
>>> reduce(pow, [1, 2, 3, 4], 2)
16777216
```



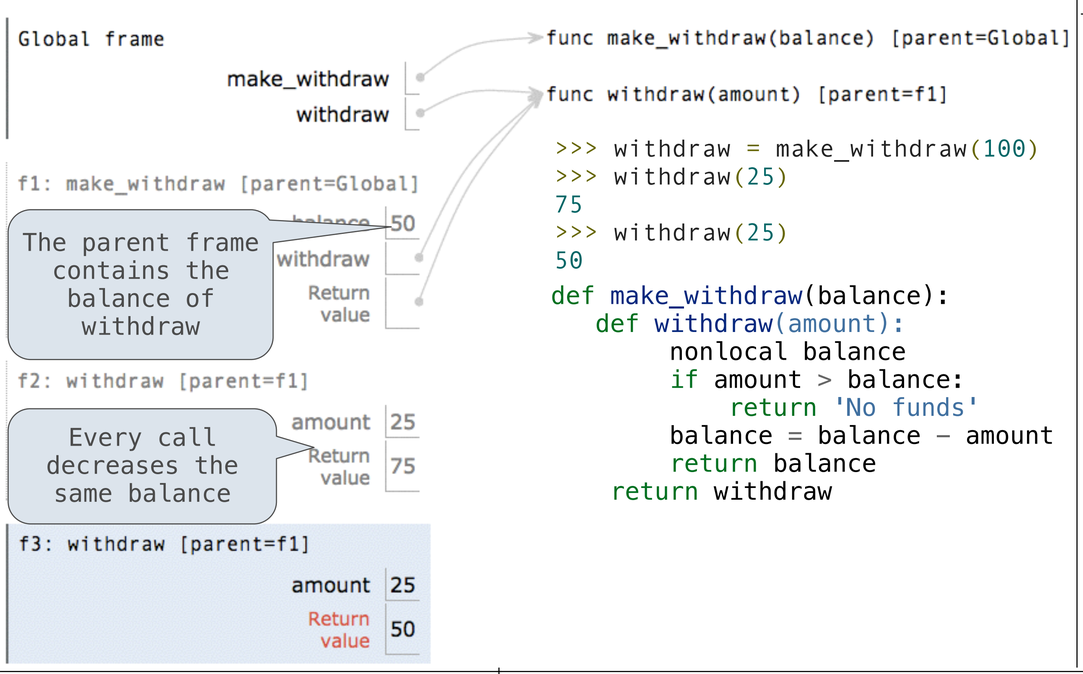
Linked list abstract data type

```
empty = ()

def link(first, rest=empty):
    return [first, rest]

def first(s):
    return s[0]

def rest(s):
    return s[1]
```



link(1, link(3, link(5)))

```
# recursive
def getitem(lst, index):
    if lst is empty:
        return 'Out of bounds'
    elif index == 0:
        return first(lst)
    return getitem(rest(lst),
                   index - 1)

# iterative
def get_item(lst, i):
    while lst != empty:
        if i == 0:
            return first(lst)
        lst, i = rest(lst), i - 1
    return 'Out of bounds'
```




```
>>> 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 and 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']
```

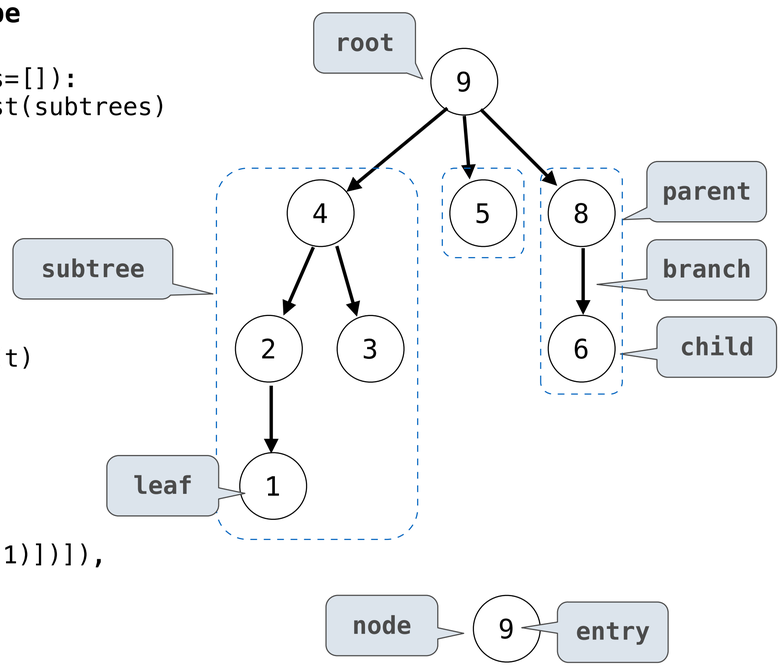
Identity:
`<exp0> is <exp1>`
evaluates to True if both `<exp0>` and `<exp1>` evaluate to the same object
Equality:
`<exp0> == <exp1>`
evaluates to True if both `<exp0>` and `<exp1>` evaluate to equal values
Identical objects are always equal values

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

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

Tree abstract data type

```
def tree(entry, subtrees=[]):
    return [entry] + list(subtrees)

def entry(t):
    return t[0]

def subtrees(t):
    return t[1:]

def is_leaf(t):
    return not subtrees(t)
```




```
tree(9,
     [tree(4,
           [tree(2),
            tree(3,
                 [tree(1)])]),
      tree(5),
      tree(8,
           [tree(6)])])
```

```
def sum_tree(t):
    if is_leaf(t):
        return entry(t)
    total = entry(t)
    for subtree in subtrees(t):
        total += sum_tree(subtree)
    return total
```

```
def map_tree(fn, t):
    if is_leaf(t):
        return tree(fn(entry(t)))
    return tree(fn(entry(t)),
                [map_tree(s) for s in subtrees(t)])
```

Linked list class (mutable)

```
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 __repr__(self):
        if self.rest is Link.empty:
            return 'Link(' + repr(self.first) + ')'
        return 'Link({0}, {1})'.format(repr(self.first), repr(self.rest))

    def __str__(self):
        elements = []
        while self is not Link.empty:
            elements.append(self.first)
            self = self.rest
        return '<' + ' '.join(elements) + '>'
```

Some zero length sequence

Sequence interface

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

repr: computer-readable representation **str**: human-readable representation

```
>>> s = Link(1, Link(2, Link(3)))
>>> s
Link(1, Link(2, Link(3)))
>>> repr(s)
'Link(1, Link(2, Link(3)))'
```

```
>>> s = Link(1, Link(2, Link(3)))
>>> print(s)
<1 2 3>
>>> str(s)
'<1 2 3>'
```

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

```
class Tree:
    def __init__(self, entry, subtrees=()):
        self.entry = entry
        for subtree in subtrees:
            assert isinstance(subtree, Tree)
        self.subtrees = list(subtree)

class BinaryTree(Tree):
    empty = ()

    def __init__(self, entry, left=empty, right=empty):
        self.entry = entry
        self.left = left
        self.right = right
```

Binary search tree: a BinaryTree that has the following qualities:
- all entries in the **left subtree** are **less than or equal** to the root's entry
- all entries in the **right subtree** are **greater than or equal** to the root's entry
- All **subtrees** are also **binary search trees**

```
def bst_contains(b, value):
    if b is BinaryTree.empty:
        return False
    elif value == b.entry:
        return True
    elif value < b.entry:
        return bst_contains(b.left, value)
    else:
        return bst_contains(b.right, value)
```

Object-Oriented Programming

Programming paradigm that thinks of code as objects, which

- have certain qualities (**attributes**)
- can perform action (**methods**)

```
class Pokemon:
    population = 0

    def __init__(self, name):
        self.name = name
        self.level = 1
        self.hp = 5 * self.level
        Pokemon.population += 1

    def level_up(self):
        self.level += 1

    def decrease_hp(self, amount):
        self.hp -= amount
        if self.hp < 0:
            self.hp = 0

    def attack(self, other):
        other.decrease_hp(self.damage)

    @property
    def damage(self):
        return 2 * self.level
```

class attribute

constructor

instance attributes

method

property method

```
>>> p = Pikachu()
>>> p.hp
5
>>> Pokemon.population
1

>>> p.decrease_hp(10)
>>> p.hp
0

>>> p.hp = 9001
>>> p.hp = 9001
9001

>>> p.damage
2
>>> p.damage = 9001
AttributeError
```

Treat property method like an attribute, but can't reassign it

```
class Pikachu(Pokemon):
    name = 'Pikachu'
    move = 'thunder'

    def __init__(self):
        Pokemon.__init__(self, Pikachu.name)

    def attack(self, other):
        print(self.name, 'used', self.move)
        Pokemon.attack(self, other)
```

Pikachu inherits from Pokemon

Pikachu overrides the `__init__` method

Calling a superclass method

<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

```
>>> p = Pikachu()
>>> p.hp
5
>>> p.move
'thunder'
>>> p.population
1
>>> p.move = 'quick attack'
>>> p.move
'quick attack'
>>> p.move
'thunder'
>>> Pokemon.population = 42
>>> p.population
42
```

Pokemon (class)
population: 0

Pikachu (subclass of Pokemon)
name: 'Pikachu'
move: 'thunder'

p (instance of Pikachu)
name: 'Pikachu'
level: 1
hp: 5
move: 'quick attack'

Iterators and generators

```
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
```