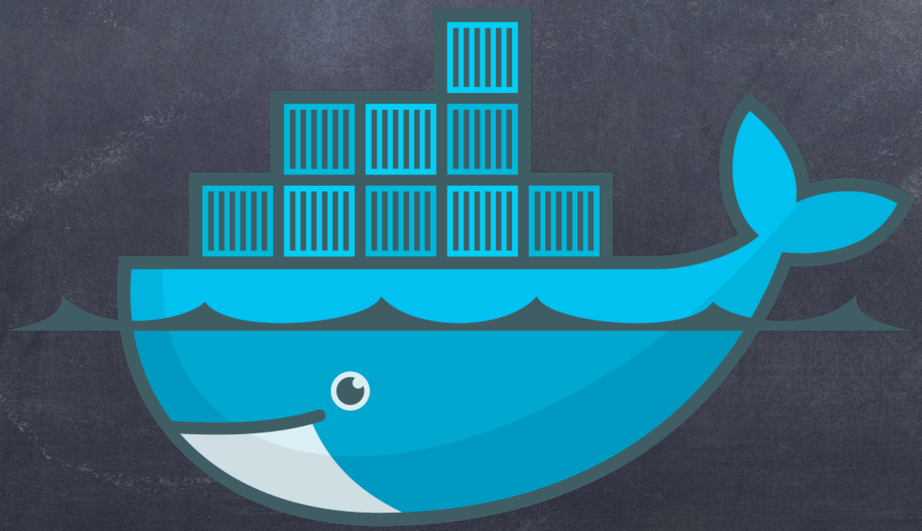
容器



列表（Lists）

- 任意数据类型的值的序列（sequence）

```
[1, 2, 3, 4, 5]

[True, "hi", 0]
```

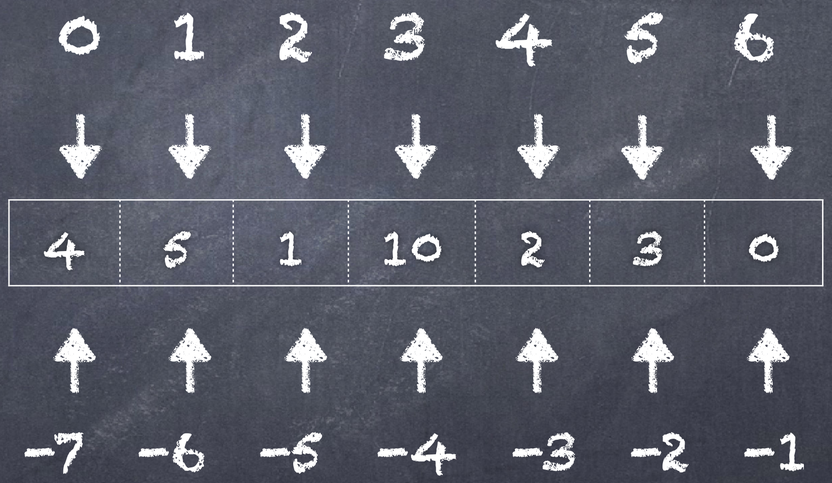
列表

- 每个在序列中的元素都有两个对应的整数指标（index）

positive index(start with 0)

```
>>> [4, 5, 1, 10, 2, 3, 0]

[4, 5, 1, 10, 2, 3, 0]
```



Negative index

列表的一些操作

```
>>> digits = [1, 8, 2, 8]
```

```
>>> digits =[2//2, 2+2+2+2, 2, 2*2*2]
```

元素的数量

```
>>> len(digits)
4
```

索引其中的元素

```
>>> digits[3]
8
```

```
>>> from operator import getitem
>>> getitem(digits, 3)
8
```

合并和重复

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

```
>> from operator import add, mul
>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

嵌套列表

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
```

```
>>> pairs[1][0]
30
```

包含关系（membership）

- 具备一个内建的操作数in，可以测试某个元素是否是一个复合数据的成员之一。

```
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```

序列迭代

- 例子：数一个列表中值等于value的个数：

```
def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total
```

名字绑定在当前环境的帧中（不是一个新的帧）

python tutor

for语句语义

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

- 首先求值头部的<expression>，其必须产生一个可迭代的值（即序列）。
- 然后对该序列中的每一个元素，依次：
  - 1.在当前帧中，将头部的<name>绑定到该元素上。
  - 2.执行<suite>中的语句。

序列解包（Sequence Unpacking）

- 可以将一个序列的各个元素分别绑定到多个名字上

```
>>> test = ['This', 'is', 'a', 'test']

>>>  a, b, c, d = test


>>> test = ['This', 'is', 'a', 'test']

>>>  head, *middle, tail = test
>>>  middle
```

序列解包（Sequence Unpacking）

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
```

Each name is bound to a value, as in multiple assignment

范围（Ranges）

范围（Ranges）

- 在一个给定范围内的一个整数的序列

```
range(<start>, <end>, <step>)
```

inclusive [optional]　exclusive　[optional]

- 当只有一个实参时，返回的是从0开始，到<end>之前结束的整数序列
- 当有两个实参时，返回的是从[start]开始，到<end>之前结束的整数序列
- 当有三个实参时，返回的是从[start]开始，到<end>之前结束的，步长为[step]的整数序列

Demo

范围（Ranges）

```
>>> range(-2, 2)  # Includes -2, but not 2
range(-2, 2)
```

可以用list构造子将一个range求值为一个list，这样list中所有的元素就和该range一样，方便查阅

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
>>> list(range(2)) # start from 0
[0, 1]
>>> list(range(2, 9, 2)) # step size is 2 instead of 1
[2, 4, 6, 8]
```

步长可以为-1吗

缺省参数

- 函数的参数缺省为默认值

缺省参数绑定该数值，缺省参数定义在必填参数之

```
>>> k_b=1.38e-23  # Boltzmann's constant
>>> def pressure(v, t, n=6.022e23):
        """Compute the pressure in pascals of an ideal
gas.

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas (default: one mole)
        """
        return n * k_b * t / v
>>> pressure(1, 273.15)
2269.974834
```

设置参数的默认值可以减少实际函数调用时设置实参的个数。

好的设计：用于函数体的大多数数据值应该表示为具名参数的默认值，这样便于查看，以及被函数调用者修改。

序列切片（Slicing）

- 获取序列的部分子序列（会创建一个新的序列）

```
s[<start index>:<end index>:<step size>]
```

inclusive [optional]

exclusive[optional]

[optional]

```
>>> s = [1, 2, 3, 4, 5, 6, 7]
>>> a = s[1:5]
>>> b = s[1::2]
>>> b = s[::]
>>> b = s[::-1]
```

Demo

Demo and Python Tutor

序列操作

- 列表推导式（List Comprehensions）

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

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

- 该复合表达式将会按照下列步骤求值一个序列：
  1. 创建一个新的帧，以当前帧为父帧
  2. 创建一个空的用于返回的结果列表
  3. 对`<iter exp>`中每一个可迭代的元素：
     - 在新的帧中将`<name>`绑定到该元素上
     - 如果`<filter exp>`求值为`True`，则将`<map exp>`的求值加入结果列表中

序列操作

- 序列聚合（Aggregation）
  - 一些内建的函数可以操作在序列的所有值上然后得到一个单一的值
  - 包括 sum, max, min, any, all

求和

最大值

最小值

至少一个元素求值为真

所有元素求值都为真

Demo

字符串（String）

字符串（String）

- 字符串是一种典型的数据抽象，由字符的序列构成，可以代表复杂的事物，比如：

表达数据

```
'200'
'1.2e-5'
'False'
'[1, 2]'
```

表达语言

```
"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

表达程序

```
'curry = lambda f:
lambda x: lambda y:
f(x, y)'
```

Demo

字符串（String）

三种形式

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

单双引号是等价的

多行字符串

\n 一个转义字符，代表换行

字符串（String）

- 字符串的操作：
  - 和List序列一样，有迭代、重复、索引、合并、查询数量等操作（语言的正交性）
  - in比较特殊，其用于查找是否是某个字串的子串

```
>>> 'here' in "Where's Waldo?"
True
```

VS

```
>>> [1, 2] in [1, 2, 3, 4]
False
```

Demo

字符串（String）

- 转化字符串函数：

```
>>> str(1)
>>> '1'
>>> str([1, 2, 3 ,4])
>>> '[1, 2, 3 ,4]'
>>> str(['1', '2'])
>>> "['1', '2']"
>>> def f(x):
...     return x
>>> str(f)
'<function f at 0x7fc909f57ca0>'
```

字典（Dictionary）

字典（Dictionary）

- 字典是键值对（Key-Value pairs）的集合

```
states = {
"CA": "California",
"DE": "Delaware",
"NY": "New York",
"TX": "Texas",
"WY": "Wyoming"
}
```

- 字典支持与List和string类似的操作

```
>>> len(states)
>>> 5
>>> "CA" in states
>>> True
>>> "ZZ" in states
>>> False
```

```
>>> "California" in states 🤔
```

字典访问

- 通过键访问值

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

```
>>> words["otro"]
'other'

>>> first_word = "agua"
>>> words[first_word]
'water'

>>> words["pavo"]
KeyError: pavo

>>> words.get("agua")
'water'

>>> words.get("pavo")
```

字典的规则

- 一个字典里的所有的键都必须是不同的
- 一个字典里的键不能是一个列表或者字典（或者其他可变mutable的类型）
- 值的类型不受限制

字典的迭代

```
>>> insects = {"spiders": 8, "centipedes": 100, "bees": 6}
>>> for name in insects:
...     print(insects[name])
...
8
100
6
```

- 键按照他们第一次加入的顺序依次迭代

字典推导式

语法：

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

Short version: {<key exp>: <value exp> for <name> in <iter exp>

语义：

按照下面的流程给出一个字典：

1. 创建一个新的帧，以当前帧为父帧
2. 创建一个空的用于返回的结果字典
3. 对<iter exp>中每一个可迭代的元素：
   - 在新的帧中将<name>绑定到该元素上
   - 如果<filter exp>求值为True，则将键值对<key exp>: <value exp>加入结果字典中

练习：Prune

```
def prune(d, keys):
    """Return a copy of D which only contains key/value pairs
    whose keys are also in KEYS.
    >>> prune({"a": 1, "b": 2, "c": 3, "d": 4}, ["a", "b", "c"])
    {'a': 1, 'b': 2, 'c': 3}
    """

    return {k: d[k] for k in keys}
```

练习：Indexing

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """

    return {k: [v for v in values if match(k, v)] for k in keys}
```

嵌套数据

Lists of lists `[ [1, 2], [3, 4] ]`

Dicts of dicts `{"name": "Brazilian Breads", "location": {"lat": 37.8, "lng": -122}}`

Dicts of lists `{"heights": [89, 97], "ages": [6, 8]}`

Lists of dicts `[{"title": "Ponyo", "year": 2009}, {"title": "Totoro", "year": 1993}]`

木（Tree）

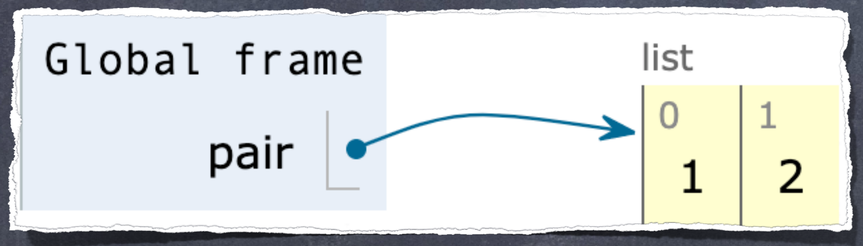
类型的闭包属性（closure property）

- 一种构成复合数据的方式具有闭包属性当且仅当：
  - 如果其生成的复合数据可以作为元素之一被用同样的方式进一步复合。
  - 比如，列表可以作为元素被另一个列表所包含。
- 闭包属性允许我们创建层次结构，即由部分组成的结构，它们本身由部分组成，等等 ⟶ 具有闭包属性是我们能够创造复合类型的关键！

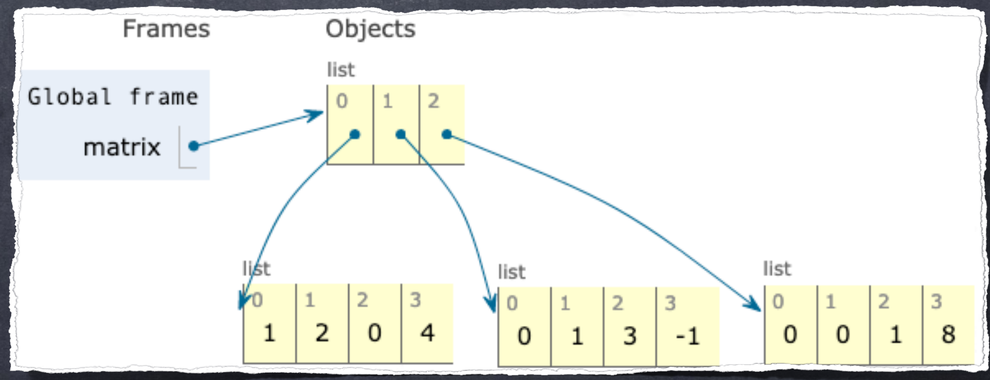
环境图中的方框指针图

- 在环境图中，列表被描述为包含列表元素的相邻框。
- 数字、字符串、布尔值和 None 等原始值出现在元素框内。复合值（例如函数值和其他列表）由箭头指示。

```
pair = [1, 2]
```

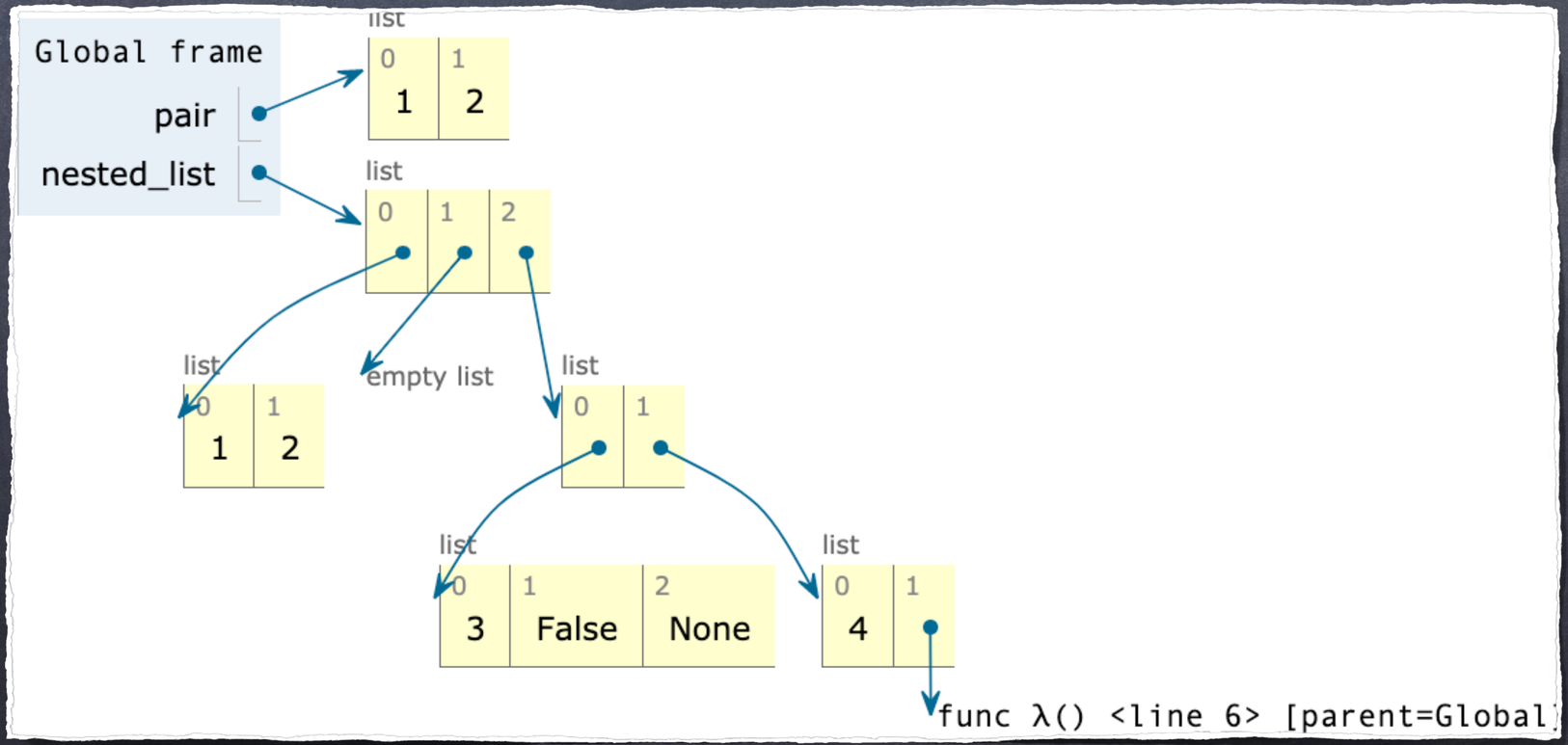



```
matrix = [ [1, 2, 0, 4], [0, 1, 3, -1], [0, 0, 1, 8] ]
```



环境图中的方框指针图

```
worst_list = [ [1, 2],
               [],
               [ [3, False, None], [4, lambda: 5]]]
```



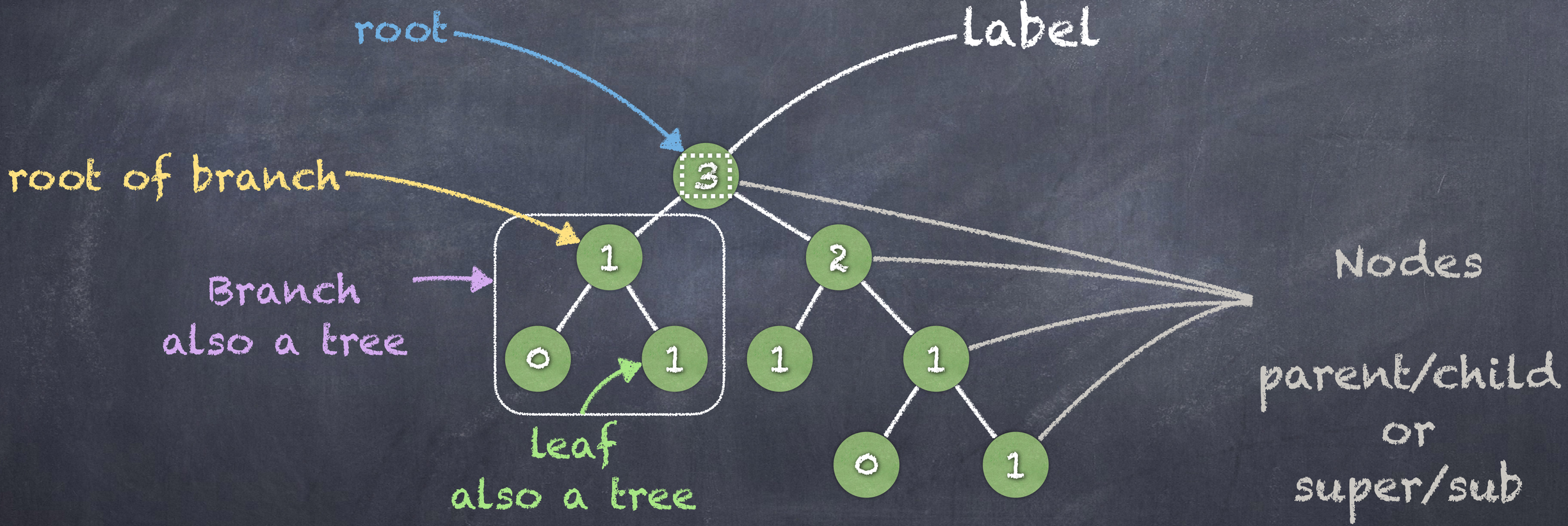
树（Tree）抽象

- 在列表中嵌套列表会带来复杂性。
- 树是一种基本的数据抽象，它对分层值的结构和操作方式施加了规律性。

树抽象

- 一棵树有一个根（root）标签和一系列分支（branches）。树的每个分支都是一棵树。没有树枝的树称为叶（leaf）。
- 树中包含的任何树都称为该树的子树（sub-tree），例如分支的分支。树的每个子树的根称为该树中的一个节点（node）。

树抽象

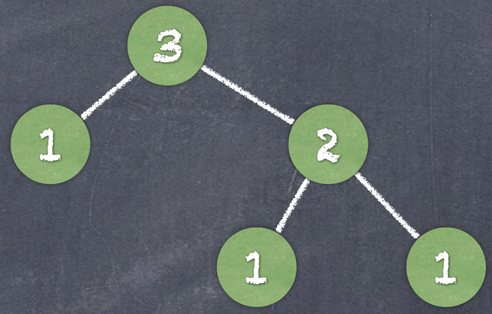

root
label
root of branch
Branch
also a tree
Leaf
also a tree
Nodes
parent/child
or
super/sub
3
1
2
0
1
1
1
0
1

实现树抽象

一个树有一个标签值
和一个分支列表

```
>>> def tree(label, branches=[]):
        return [label] + branches)
```

```
>>> def label(tree):
        return tree[0]
>>> def branches(tree):
        return tree[1:]
```



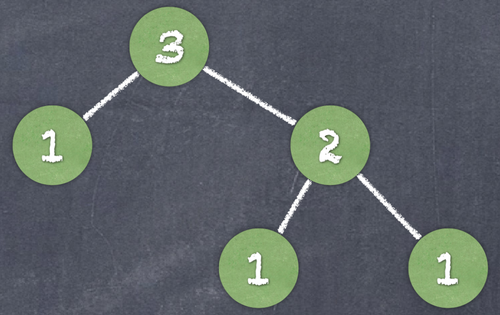

```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])

[3, [1], [2, [1], [1]]]
```

Demo

实现树抽象

一个树有一个标签值
和一个分支列表




```
>>> 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
```

每一个分支都是一个树

创建branches的list

类型是list吗？

```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

```
>>> def is_leaf(tree):
        return not branches(tree)
```

树操作

- 可以利用树递归函数来处理树
- 例子：计算叶子树

```
def count_leaves(t):
    """Count the leaves of a tree."""
```

Demo

树操作

- 一般处理叶子结点是树操作的基本情形
- 递归部分一般是对每个分支进行递归调用，然后最后聚合结果

```
def count_leaves(t):
    """Count the leaves of a tree."""
    if is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)
```

树操作

- 可以利用树递归函数来创造树
- 例子：例如构造 Fibonacci 树（Fibonacci的根标签是第 n 个 Fibonacci 数，对n > 1，两个分支也是 Fibonacci 树）.

```
def fib_tree(n):
    """Construct the Fibonacci tree."""
```

树操作

基本情况

更小问题

解决更大问题

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

>>> count_leaves(fib_tree(5))
8
```

树操作

- 另一个例子：Partition Tree

链表（Linked List）

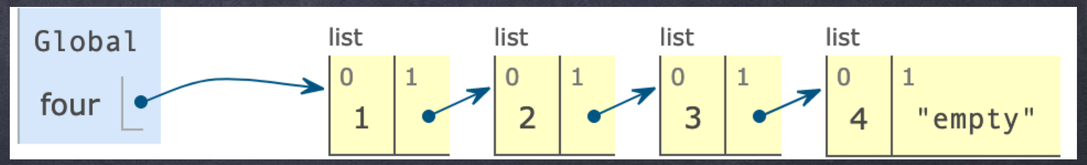
链表（Linked List）

- python的list是一个"动态数组"，虽然已经非常有用，但是在一些情况下不是最优的实现
  - 比如插入元素（特别是靠近前端的插入）
  - 而且插入元素过多，超过了之前的申请空间，还需要涉及重新创建数组

链表（Linked List）

- 链表是一个对象的链，链里面每一个对象都持有一个值和指向下一个对象的链接。
- 链表的最末对象的指向是空。

```
four = [1, [2, [3, [4, 'empty']]]]
```

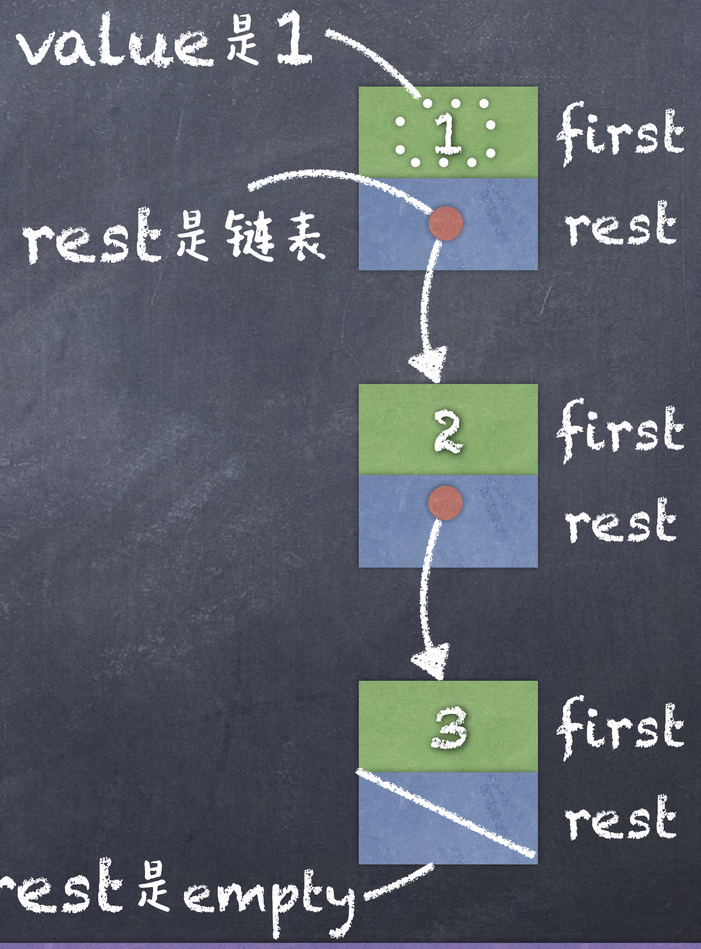


链表（Linked List）

- 链表具有递归结构：链表的其余部分是链表或Empty.
- 我们可以定义一个抽象数据表示来验证、构造和选择链表的组件。

链表（Linked List）

```
>>> empty = 'empty'
>>> def is_link(s):
        """s is a linked list if it is empty or a (first, rest) pair."""
        return s == empty or (len(s) == 2 and is_link(s[1]))
>>> def link(first, rest):
        """Construct a linked list from its first element and the rest."""
        assert is_link(rest), "rest must be a linked list."
        return [first, rest]
>>> def first(s):
        """Return the first element of a linked list s."""
        assert is_link(s), "first only applies to linked lists."
        assert s != empty, "empty linked list has no first element."
        return s[0]
>>> def rest(s):
        """Return the rest of the elements of a linked list s."""
        assert is_link(s), "rest only applies to linked lists."
        assert s != empty, "empty linked list has no rest."
        return s[1]
```



链表可以看成是value的序列

链表（Linked List）

- 上述代码中，link是构造子，first和rest是选择子

```
>>> four = link(1, link(2, link(3, link(4, empty))))
>>> first(four)
1
>>> rest(four)
[2, [3, [4, 'empty']]]
```

根据数据抽象，除了使用list之外，我们可以用其它方式（比如高阶函数）实现构造子和选择子

链表（Linked List）

- 满足序列抽象：长度和元素选择

```
>>> def len_link(s):
        """Return the length of linked list s."""
        length = 0
        while s != empty:
            s, length = rest(s), length + 1
        return length

>>> def getitem_link(s, i):
        """Return the element at index i of linked list s."""
        while i > 0:
            s, i = rest(s), i - 1
        return first(s)
```

```
>>> len_link(four)
4
>>> getitem_link(four, 1)
2
```

python tutor

链表（Linked List）

- 上述过程可以利用递归实现

```
>>> def len_link_recursive(s):
        """Return the length of a linked list s."""
        if s == empty:
            return 0
        return 1 + len_link_recursive(rest(s))
>>> def getitem_link_recursive(s, i):
        """Return the element at index i of linked list s."""
        if i == 0:
            return first(s)
        return getitem_link_recursive(rest(s), i - 1)
>>> len_link_recursive(four)
4
>>> getitem_link_recursive(four, 1)
2
```

链表（Linked List）

- 利用递归可以很容易做到如链表的结合：

```
>>> def extend_link(s, t):
        """Return a list with the elements of s followed by those of
        assert is_link(s) and is_link(t)
        if s == empty:
            return t
        else:
            return link(first(s), extend_link(rest(s), t))

>>> extend_link(four, four)
[1, [2, [3, [4, [1, [2, [3, [4, 'empty']]]]]]]]
```

链表（Linked List）

- 利用递归可以很容易做到如链表的操作（类似推导式）：

```
>>> def apply_to_all_link(f, s):
        """Apply f to each element of s."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            return link(f(first(s)), apply_to_all_link(f, rest(s)))
>>> apply_to_all_link(lambda x: x*x, four)
[1, [4, [9, [16, 'empty']]]]
```

链表（Linked List）

- 利用递归可以很容易做到如链表的操作（类似推导式）：

```
>>> def keep_if_link(f, s):
        """Return a list with elements of s for which f(e) is true."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            kept = keep_if_link(f, rest(s))
            if f(first(s)):
                return link(first(s), kept)
            else:
                return kept
>>> keep_if_link(lambda x: x%2 == 0, four)
[2, [4, 'empty']]
```

更多有关Python的详细语法可以参照文档查阅

https://docs.python.org/3/library/stdtypes.html

https://docs.python.org/3/tutorial/index.html

Any questions ?