可变性

元组（Tuple）

- 元组也是一种python内建的序列，其一旦创建，后续无法改变其值
  - 以逗号分隔元素表达式的元组文本创建的，圆括号是可选的（但在实践中经常使用）
  - 和列表一样，元组内的元素可以为任意类型的对象

元组

```
>>> 1, 2 + 3
(1, 5)
>>> ("the", 1, ("and", "only"))
('the', 1, ('and', 'only'))
>>> type( (10, 20) )
<class 'tuple'>

>>> ()     # 0 elements
()
>>> (10,) # 1 element
(10,)
```

闭包属性

空元组，唯一不需要逗号的元组

单个元素元组（注意逗号） - it is the comma, not the parentheses, that define the tuple.

元组的操作

- 和列表一样，我们可以查询元组中元素的数量、索引其中的元素等

```
>>> code = ("up", "up", "down", "down") + ("left", "right") * 2
>>> len(code)
8
>>> code[3]
'down'
>>> code.index("left")
4
>>> code.count("down")
2
```

元组的操作

- 但不能更改其中的元素

```
>>> a = (1, 5)
>>> a[0] = 1
```

```
>>> b = (1, [1, 2, 3])
>>> b[1] = [2, 3]
```

TypeError: 'tuple' object does not support item assignment

- 但其某个元素如果是可变的，那么该值可以变化

```
>>> b = (1, [1, 2, 3])
>>> del b[1][0]
>>> b
(1, [2, 3])
```

python tutor

可变性（Mutability）

可变性

⚠这里的思想与函数式编程不同！

- 程序的运行可以看成是一些对象的值的变化过程
- 如果能组织和维护这些变化显然有助于构建复杂的程序

不可变性vs可变性（Immutability vs Mutability）

一个不可改变的值当其创建之后即不可更改，如：int, bool, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3
a_string = "Hi y'all"
a_string[1] = "I"
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

Error

Error

🤔这是怎么做到的？

🤔这是怎么做到的？

tips: id()

一个可改变的值当其创建之后可以随程序的运行改变值，如：list, dictionary

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey
```

demo

函数调用中的可变性

- 函数可以改变其所能访问到的域里的对象的值（其本身是可变的）

```
four = [1, 2, 3, 4]
print(four[0])
do_stuff_to(four)
print(four[0])
```

- 即使不传实参也能改变

```
four = [1, 2, 3, 4]
print(four[3])
do_other_stuff()
print(four[3])
```

python tutor

回顾树

```
def tree(label, branches=[]):
    """ Creates a tree whose root node is labeled LABEL and
        optionally has CHILDREN, a list of trees."""
    return ([label] + list(children))

def label(tree):
    """ Returns the label of the root node of TREE. """
    return tree[0]

def branches(tree):
    """ Returns a list of children of TREE. """
    return tree[1:]
```

树是可变的吗？

不可变！一旦创建，我们没有改变其值的操作

（不违背抽象界限的情况下）

可改变的树

- 添加如下代码：

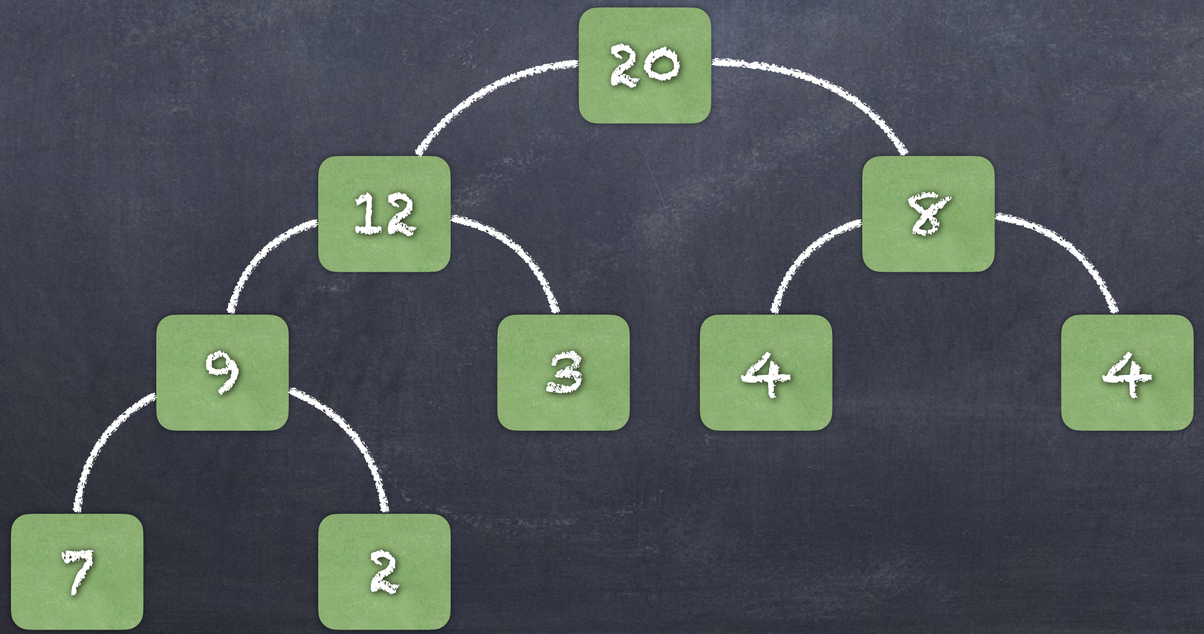
```
def set_label(tree, label):
    """Sets the label of TREE's root node to LABEL"""
    tree[0] = label

def set_children(tree, children):
    """Sets the children of TREE to CHILDREN, a list of trees."""
    tree[1:] = children
```

变异子(mutator)

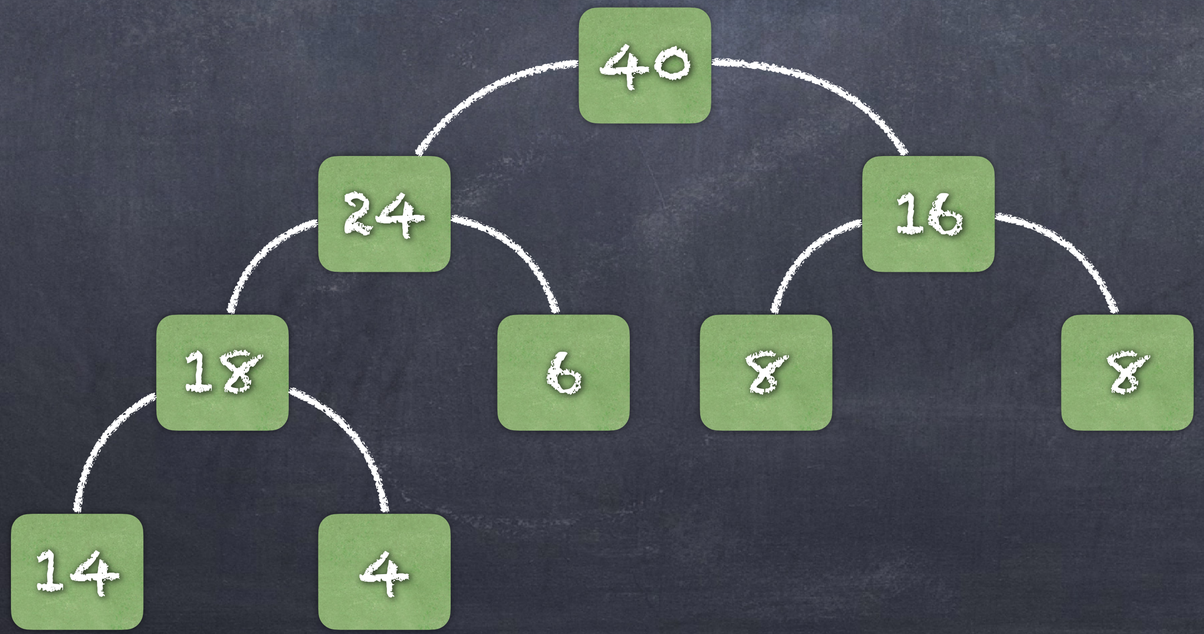
例子：改变树

```
def double(t):
    """Doubles every label in T, mutating T."""
    set_label(t, label(t) * 2)
    if not is_leaf(t):
        for c in branches(t):
            double(c)
```



例子：改变树

```
def double(t):
    """Doubles every label in T, mutating T."""
    set_label(t, label(t) * 2)
    if not is_leaf(t):
        for c in branches(t):
            double(c)
```



破坏性 vs 非破坏性（Destructive and Non-destructive）

- 对于可变的数据而言，并不是所有操作都会改变其数据
  - **破坏性**：该操作会改变原来的数据
  - **非破坏性**：该操作不会改变原来的数据（可能会创建新的数据）

再次回顾列表

- 列表是可变的
- 但并不是所有的操作都改变列表

```
listA = [2, 3]
listC = listA[:]
listC[0] = 4
```

slicing

```
listA = [2, 3]
listC = list(listA)
listC[0] = 4
```

创建一个新的列表

```
listA = [2, 3]
listC =listA.copy()
listC[0] = 4
```

拷贝

demo

拷贝

- 浅拷贝（shallow copy）
  - 创建一个新的对象，然后把原对象中的元素的引用拷贝进去
- 深拷贝（deep copy）
  - 创建一个新的对象，然后递归地把原对象中的所有元素拷贝进去

拷贝

```
import copy
a = [1, 2, 3, 4, ['a', 'b']]
b = a
c = copy.copy(a)
e = a.copy()
d = copy.deepcopy(a)
a.append(5)
a[4].append('c')
a
b
c
d
e
```

赋值，绑定了相同的对象（即引用）

浅拷贝，把内部元素引用进行拷贝，不涉及更深的元素

深拷贝，把内部元素进行递归地拷贝（所有可能的元素）

demo

一些对列表的破坏性操作

- 通过[ ]和slicing来进行破坏性操作

```
L = [1, 2, 3, 4, 5]
L[2] = 6
L[1:3] = [9, 8]
L[2:4] = []              # Deleting elements
L[1:1] = [2, 3, 4, 5]    # Inserting elements
L[len(L):] = [10, 11]    # Appending
L = L + [20, 30]
L[0:0] = range(-3, 0)    # Prepending
```

一些对列表的破坏性操作

- 通过append和extend来进行破坏性操作

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

```
s = [2, 3]
t = [5, 6]
s.extend(t)
t = 0
```

python tutor

一些对列表的破坏性操作

- 通过pop和remove来进行破坏性操作

```
s = [2, 3]
t = [5, 6]
t = s.pop()
```

```
s = [6, 2, 4, 8, 4]
s.remove(4)
```

python tutor

字典是当然也可以进行破坏性的操作

- 字典的一些破坏性操作

```
a = {"1":3, "4": 5}
a["4"] = 6 #更新信息
a[5] = [1, 2, 3]  # 增加元素为：{'1': 3, '4': 6, 5: [1, 2, 3]}
del a["1"] # 删除键 "1"和其相应的值
a.clear()  # 清空所有的数据
a = {'4': 6, 5: [1, 2, 3]}
del a #删除字典
```

字典是当然也可以进行破坏性的操作

- 其他一些常见的字典的操作

```
a = {"1":3, "4": 5}
b = {"1":5, 7: [1, 2, 3]}
a.update(b) #将b中的元素添加到a中，如果有相同的key，更新key的值
c = a.pop("1")
```

同一性（Identity）vs 相等性（Equality）

同一对象vs内容相等

（Identity of objects vs. Equality of contents）

- 同一性：两个数据是同一的当且仅当他们指向同一个对象（id相同）
  - `<exp0> is <exp1>`
  - 返回True如果exp0和exp1两个求值的对象是同一对象
- 相等性：两个数据是相等的当且仅当他们所蕴含的"值"是相等的（可以是不同的对象）。
  - `<exp0> == <exp1>`
  - 返回True如果exp0和exp1两个求值的对象所含的值是相等的。

同一性 vs 相等性

```
a = ["apples", "bananas"]
b = ["apples", "bananas"]
c = a
print(a is b)

if a == b == c:
    print("All equal!")

a[1] = "oranges"

if a is c and a == c:
    print("A and C are equal AND identical!")

if a == b:
    print("A and B are equal!") # Nope!

if b == c:
    print("B and C are equal!") # Nope!
```

Identical objects always have equal values.

python tutor

不可变数据中的"同一性"

```
a = "orange"
b = "orange"
c = "o" + "range"
print(a is b)
print(a is c)
```

Beware: may not act like you expect for strings/numbers!
Python interpreter does some optimization for strings/numbers

```
a = 100
b = 100
print(a is b)
print(a is 10 * 10)
print(a == 10 * 10)

a = 500
b = 500
print(a is b)
print(500 is 500)
```

demo

作用域（Scope）

在局部（Local）作用域的名字

可以正常运行吗？

```
attendees = []

def mark_attendance(name):
    attendees.append(name)
    print("In attendance:", attendees)

mark_attendance("Emily")
mark_attendance("Cristiano")
mark_attendance("Samantha")
```

可以正常运行吗？

```
current = 0

def count():
    current = current + 1
    print("Count:", current)

count()
count()
```

UnboundLocalError: local variable 'current' referenced before assignment

作用域规则

动作	全局帧的代码	本地帧的代码
可以访问全局帧里绑定的名字？	可以	可以
对全局帧绑定的名字进行重新赋值？	可以	不可以（除非申明 global）

python tutor

全局变量重新赋值

```
current = 0

def count():
    global current
    current = current + 1
    print("Count:", current)

count()
count()
```

python tutor

尽量避免使用global

在函数本地对全局变量的重新赋值可能会导致写出的代码是不够健壮的，并且逻辑难以预测

```
current = 0

def count(current):
    current = current + 1
    print("Count:", current)
    return current

current = count(current)
current = count(current)
```

用参数代替

在嵌套的作用域里的名字

可以正常运行吗？

```
def make_tracker(class_name):
    attendees = []

    def track_attendance(name):
        attendees.append(name)
        print(class_name, ": ", attendees)

    return track_attendance

tracker = make_tracker("SE-Computation1")
tracker("Emily")
tracker("Cristiano")
tracker("Julian")
```

可以正常运行吗？

```
def make_counter(start):
    current = start

    def count():
        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

UnboundLocalError: local variable 'current' referenced before assignment

作用域规则

访问绑定在enclosing function里的名字？	可以
重新赋值绑定在enclosing function的名字？	不可以（除非申明 nonlocal）

python tutor

父作用域的名字重新赋值

```
def make_counter(start):
    current = start

    def count():
        nonlocal current
        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

The nonlocal declaration tells Python to look in the parent frame for the name lookup

python tutor

同样，应避免使用nonlocal

We could use a mutable value like a list or dict:

```
def make_counter(start):
    current = [start]

    def count():
        current[0] += 1
        print("Count:", current[0])

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

```
current = current
current[0] +=1
```

python tutor

nonlocal的另一个作用

可以利用nonlocal来构造可变的数据抽象

```
def pair(a, b):
    def pair_func(which, v=None):
        nonlocal a, b
        if which == 0:
            return a
        elif which == 1:
            return b
        elif which == 2:
            a = v
        else:
            b = v
    return pair_func
```

```
def left(p):
    return p(0)

def right(p):
    return p(1)

def set_left(p, v):
    p(2, v)

def set_right(p, v):
    p(3, v)
```

```
aPair = pair(3, 2)
set_left(aPair, 5)
print(left(aPair))
```

利用List避免nonlocal

```
def pair(a, b):
    return [a, b]

def left(p):
    return p[0]

def right(p):
    return p[1)

def set_left(p, v):
    p[0] = v

def set_right(p, v):
    p[1] = v
```

```
aPair = pair(3, 2)
set_left(aPair, 5)
print(left(aPair))
```

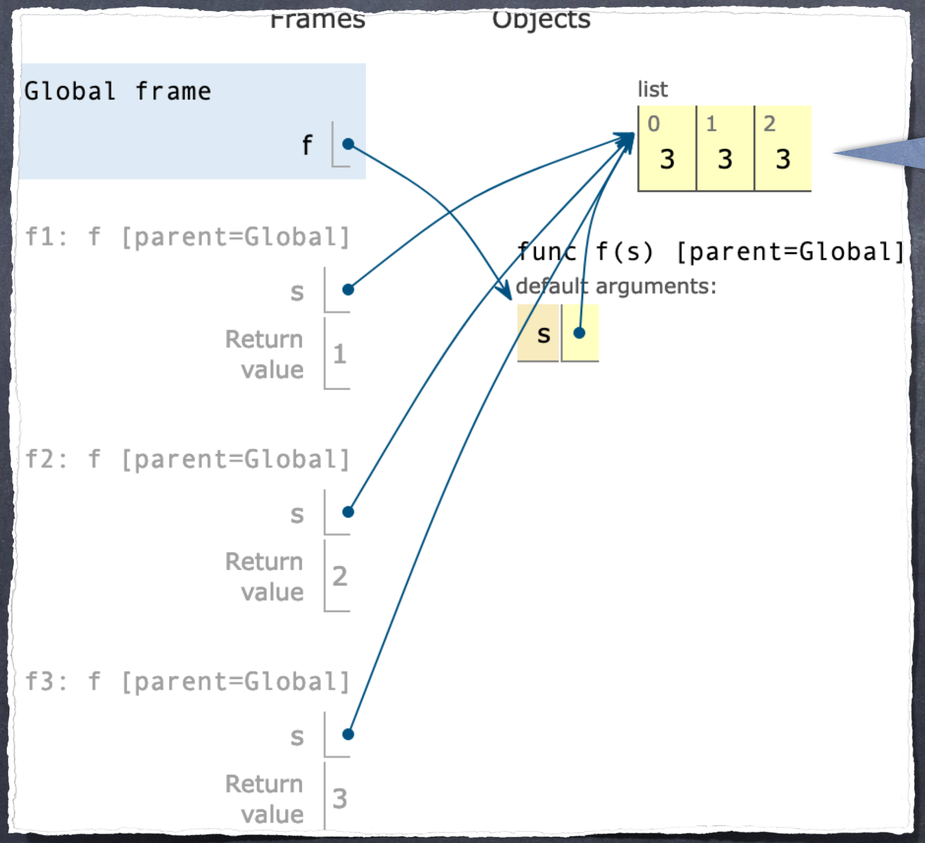
作用域规则

动作	全局帧的代码	本地帧的代码	嵌套函数里的代码
可以访问全局帧里绑定的名字？	可以	可以	可以
对全局帧绑定的名字进行重新赋值？	可以	不可以（除非申明 global）	不可以（除非申明global）
访问绑定在enclosing function里的名字？	N/A	N/A	可以
重新赋值绑定在enclosing function的	N/A	N/A	不可以（除非申明nonlocal）

可变的缺省参数是危险的

- 缺省的实参是函数值的一部分，不是调用时才产生的

```
>>> def f(s=[]):
...     s.append(3)
...     return len(s)
...
>>> f()
1
>>> f()
2
>>> f()
3
```



每次（缺省）调用都绑定在同样的对象上

python tutor

Any questions ?