CS61A: Structure and Interpretation of Computer Programs

Created	@July 20, 2022 7:38 PM
Class	CS61A 20Fall
Type	编程入门
Finished	☐

Useful Links:

- 2020Fall课程地址：https://inst.eecs.berkeley.edu/~cs61a/fa20/
- Python Frames and Binding Tutor: https://pythontutor.com/composingprograms.html#mode=edit
- TextBook: http://composingprograms.com/

Week1

Week1

Lecture 2: Functions

Names, Assignments, and User Defined Functions

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b75c09a8-7996-4d9e-bba7-a0a1d9e9ba14/02-Functions_1pp.pdf

python里的变量函数等都可以给它们起一个别名

```
>>> f = max
>>> max = 10
>>> f(1, 2, max)
10
```

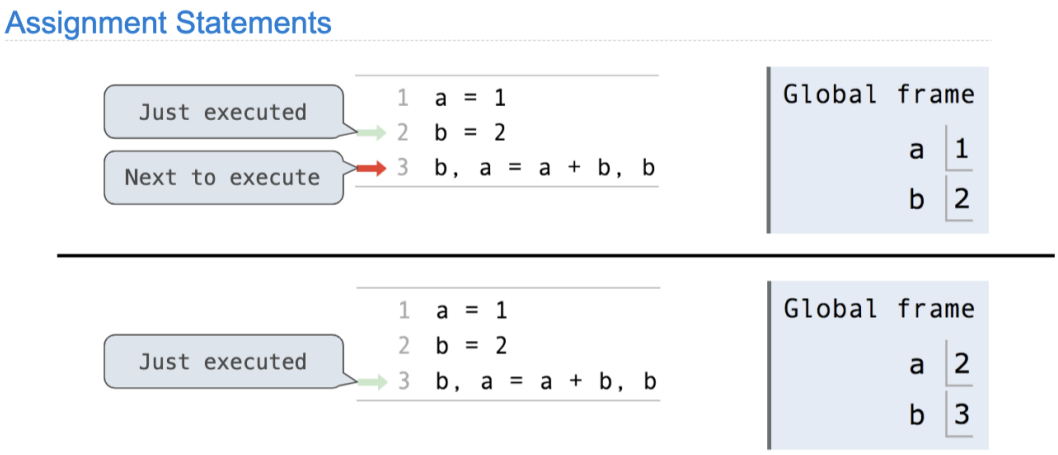
使用def来定义Python里的函数，注意缩进

```
>>> def square(x):
...     return x * x
...
>>> square(2)
4
```

⚠️ Python里的函数的参数就是我们上文中所说的“别名”，也就是names，所以函数，变量都可以作为函数的参数

Environments Diagrams and Defining Functions

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2749f80a-fdd3-457d-85f6-89e006ed5089/02-Functions_1pp.pdf



可以使用Frame这种形式来记录name和值的绑定关系，有助于程序员分析

执行def语句时只是把函数名和整个函数体（object）绑定

当你调用函数的时候，会创建一个新的local frame，相当于一个新的空间，建立新的local frame的时候需要利用到函数签名（function's signature），函数签名所含的信息就已经完全足够建立local frame了

⚠️ 重要的内容：
1. 一个环境就是一系列的有序的frames组成的，local frame早于global frame
2. 一个name等价于在当前环境中最早的那个frame中所找到的同名name所绑定的值

Week2

Lecture 3: Control

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/66625ae5-7089-4b6b-affd-66bc9b9127d0/03-Control_1pp.pdf

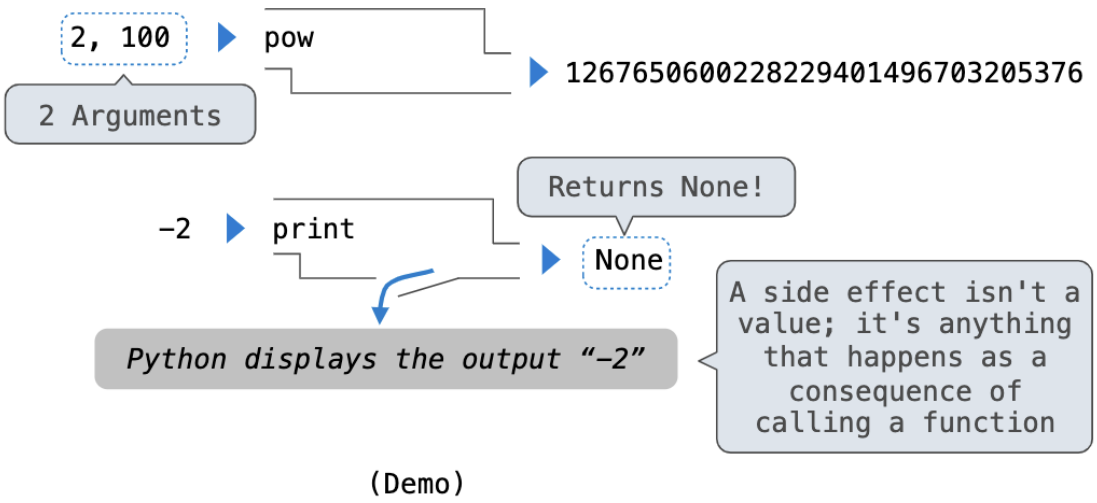
Print and None

我们先看个小例子

```
>>> -2
-2
>>> print(-2)
-2
>>> None
>>> print(None)
None
>>>>>> print(print(1), print(2))
1
2
None None
```

None 是Python的一个关键词，代表啥都没有

⚠️ 没有return的函数默认返回None，Python的解释器不会把None这个返回值自动打印出来（也就是如果你不写print的话，不会自动打印的）



(Demo)

没有返回值的函数也可以干别的，比如print就是做了一个打印到屏幕的事情，然后返回None

这样我们就可以解释例子了：执行print(1), print(2)的时候得到（1 \n 2）；最外面的print输出的是print(1), print(2)的返回值None

⚠️ 所以我们可以认为有两种函数：带返回值的函数 和 没有返回值的函数（返回值是None）

Miscellaneous Python Features

常规运算符

- 加法： a+b
- 减法： a-b

- 乘法： a*b
- 括号： ()
- 除法： a/b
- 真除法（整数除法）： a // b
- 取模： a%b

Python的执行方式：

1. 交互式： 直接开控制台来输入命令 `>>>python`
2. 解释式： 写成文件之后再解释执行 `>>>python3 ex.py`

   解释式的附加参数：

   1. 执行后再进入交互模式： 执行文件后再进入交互形式 `>>>python3 -i ex.py`
   2. 测试文件：执行函数下方的描述字符串里的测试用例，例子如下 `>>>python3 -m doctest -v ex.py`

python 一般在函数`def` 语句下的第一行写函数的文档，以及测试用例，如：

```
def divide_exact(n, d):
  """Return the quotient and remainder of dividing N by D.
     再写个例子：
     >>> q, r = divide_exact(2013, 10)
     >>> q, r
     (201, 3)

  """
  return n // d, n % d
```

测试样例：

```
(base) ➜  Division git:(master) ✗ python3 -m doctest -v ex.py
quotientn is : 202
remiander is : 2
Trying:
    q, r = divide_exact(2013, 10)
Expecting nothing
ok
Trying:
    q, r
Expecting:
    (201, 3)
ok
1 items had no tests:
    ex
1 items passed all tests:
   2 tests in ex.divide_exact
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
(base) ➜  Division git:(master) ✗
```

python也可以设置默认参数（default arguments）,如下：

```
def divide_exact(n=2013, d=10):
```

Conditional Statements

基本的条件控制语句

```
def absolute_value(x):
  if x < 0:
    return -x;
  elif x == 0:
    return 0;
  else:
    return x;
```

Iteration

while循环

```
i, total =0, 0
while i<3:
  i  = i+1
  total = total + i
```

Lecture 4: Higher-Order Functions

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d5a9af1c-5dc7-445d-9561-4fb3f7a5c65d/04-Higher-Order_Functions_1pp.pdf

Designing Functions

以前是没有函数的，这样就是一大堆的statements互相跳转，很乱。函数式编程可以改善这种情况，所以学习怎么写出好的函数是很重要的。

⚠️ 和数学里的函数类似，程序里的函数也需要注意以下这些特征：

1. 函数的作用域（domain）：输入（也就是函数参数的取值范围）的取值范围
2. 函数的值域（range）：return语句返回的范围
3. 函数的行为（behavior）：输入和输出之间的联系

Generalizing Patterns with Arguments

断言表达式 `assert <bool expression>, <Error info String>`

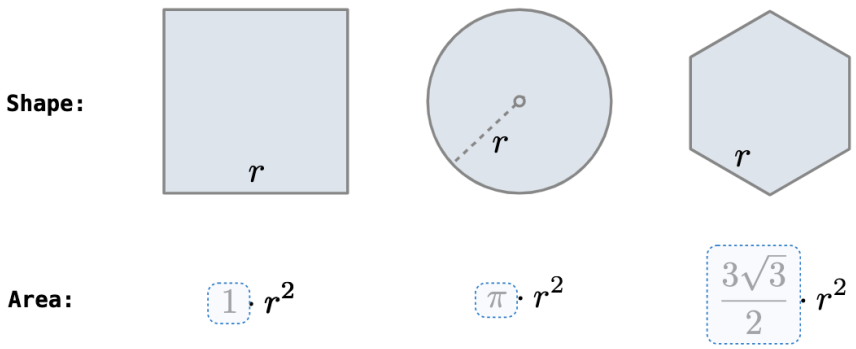
断言表达式的作用一般是错误控制

例子：

```
  ex.py 内容如下
  1 from math import pi
  2
  3 def circle_area(r):
  4     assert r > 0, 'A length must be positive!'
  5     return r * r * pi
  6


(base) ➜  Higher-OrderFunctions git:(master) ✗ python3 -i ex.py
>>> circle_area(4)
50.26548245743669
>>> circle_area(-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/lanbangxiang/StudyStuff/CS自学指南/CS61A/CoursePractice/Week1/Higher-OrderFu
nctions/ex.py", line 4, in circle_area
    assert r > 0, 'A length must be positive!'
AssertionError: A length must be positive!
>>>
```

函数的抽象： 我们写程序的时候要注意函数抽象的层级，最好写出多层级不同抽象度的函数。如下图中的计算面积和计算和都可以抽象出两个层级来。

Regular geometric shapes relate length and area.



Finding common structure allows for shared implementation

(Demo)

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^{5} \frac{8}{(4k-3)\cdot(4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

(Demo)

Functions as Return Values

返回值为函数的函数

例子如下：

```
 1 def maker_adder(n):
 2     """
 3         返回一个以k为参数，返回k + n的函数
 4     """
 5     def adder(k):
 6         return k + n;
 7 
 8     return adder
 9 
10 


(base) ➜  Higher-OrderFunctions git:(master) ✗ python3 -i ex1.py
>>> f3 = maker_adder(3)
>>> f3(3)
6
>>>
```

⚠️ 重点

- **函数在何时执行？** 注意执行到def语句时只是绑定name，并不执行函数体的内容（也就是说这时没有传参），只有你真正的call这个函数的时候才会传参，函数体才会被执行
- **Functions are first-class**：函数是一等公民，在Python中函数可以像变量一样操作
- **Higher-Order Function**：高阶函数可以接受别的函数作为参数，也可以返回函数

High-Order Function的作用：

1. 表达计算的通用方法
2. 减少程序中的重复
3. 细分函数，使得每个函数就只干一件事

Lambda Expression

一行就能写下的函数

格式为：`lambda <paras>: <return expression>`

如：`lambda x, y: x * x + y * y`

lambda expression 和 def 的区别就是 def 会给出那个函数一个固有的名字，而 lambda 没有

Return

return语句执行之后就会回到之前的环境当中,相应的return也为原来的环境引入了新的信息
return语句只能存在于函数体中。

这里给一个high order function 的例子，inverse也就是参数是函数的函数。

```
def search(f):
      """
          这是high order function, 接受的参数f是一个函数的别名
      """
      x = 0
      while f(x) == False:
          x += 1
      return x

  def square(x):
      return x * x

  def positive(x):
     return max(0, square(x)-100)


  def inverse(f):
      """
      return g(y) such that g(f(x)) -> x
      """

      return lambda y: search(lambda x: f(x) == y )
```

Control

只有call 语句（Python里也就是函数）是不够的。必须还有if, while 等条件控制语句。

Control Expression

逻辑操作符

- and 与
- or 或

值得注意的是：

- Python在计算 <op> and <op> 表达式的时候当遇到第一个False的<op>时会立刻返回
- Python在计算 <op> or <op> 表达式的时候当遇到一个True的<op>时会立刻返回

例子：

```
>>> 1 or False
1
>>> 0 and True
0
```

if语句的缩写形式

```
 <consequent> if <predicate> else <alternative>
```

等价于：

```
if predicate:
   consequent
else:
   alternative
```

例子如下：

```
>>> x=0
>>> abs(1/x if x != 0 else 0)
0
```

Lecture 5: Environments

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d9cf1abf-6ed6-4b43-acb6-cb4193bf4523/05-Environments_1pp.pdf

Environments for Higher-Order Functions

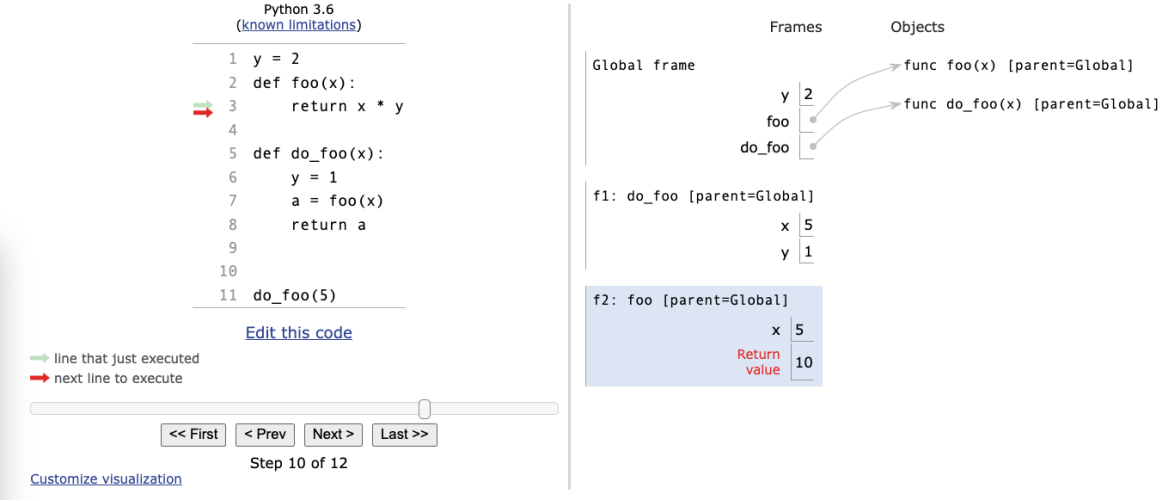
高阶函数定义：可以接受函数形式的参数，或返回值为函数形式的函数就叫高阶函数

⚠️ 一个函数完整的执行过程：

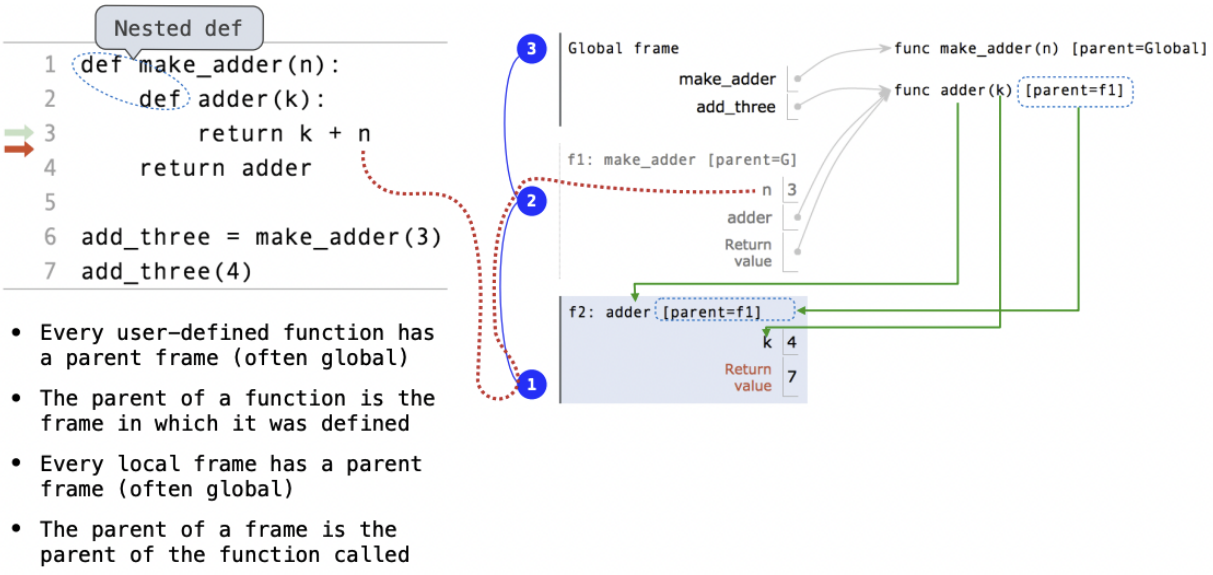
1. 执行def语句： 给这个函数绑定一个别名，知道形参是什么，知道这个函数的parent frame是什么
2. 执行到其他的语句： blah blah blah，无事发生
3. 当你调用（call）这个函数的时候
    a. 创建新的frame
    b. 传参数
    c. 执行函数体，return结束函数

找名字被bound到谁的时候，顺序为从local frame 往外找

💡 注意函数的parent frame是其被定义的时候所在的frame, 这叫做lexical scope。例子如下：
更多信息可以参考：
https://prl.ccs.neu.edu/blog/2019/09/05/lexical-and-dynamic-scope/



Environments for Nested Definitions



如上图，def里有新的def就叫做Nested Definitions

注意看右边的帧，每个帧都记录着它们的上一级的environment是什么，这样才知道每个帧的顺序。这样在找名字的时候就会先在当前这层去找，如果没找到，就往上层的帧迭代下去找，直到找到相应的值

Local Names

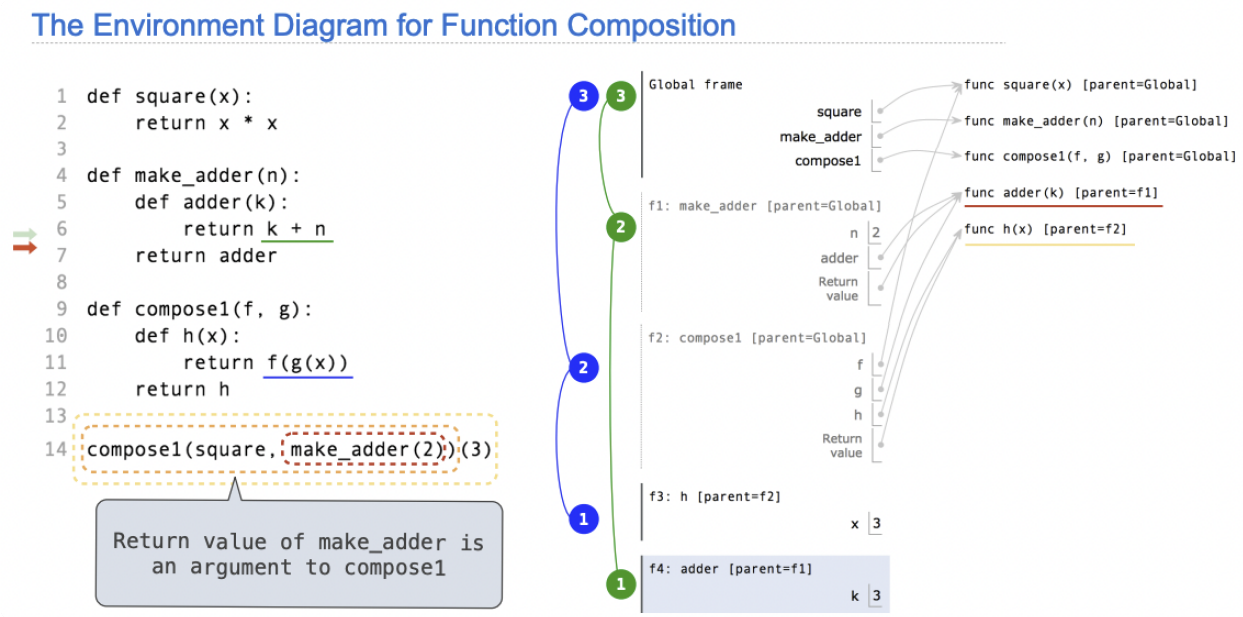
简单介绍了一下局部变量

例子：

```
>>> def f(x, y):
...     return g(x)
```

```
...
>>> def g(a):
...     return a + y
...
>>> result = f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in g
NameError: name 'y' is not defined
>>>
```

Function Composition



调用顺序：square → 绑定square函数 → make_adder() → 得到adder → 绑定adder → compose1() → 计算出结果

Self-Reference

Python支持在函数里调用自己这种方式，叫做递归

Function Currying

Currying: 把一个多参数的函数变成一个多层嵌套的单参数的高阶函数。

例子如下：

```
  1 def curry2(f):
  2     def g(x):
  3         def h(y):
  4             return f(x, y)
  5         return h
  6     return g



(base) ➜  CoursePractice git:(master) ✗ python3 -i curry.py
>>> from operator import add
>>> m = curry2(add)
>>> add3 = m(3)
>>> add3(4)
7
>>>
```

Week 3

Lecture 6: Design

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b60e51bf-f3b7-4c94-9d71-855191ddaa8f/06-Design_1pp.pdf

Abstraction

函数的抽象：当你去调用一个函数的时候，你需要知道这个函数的什么内容呢？

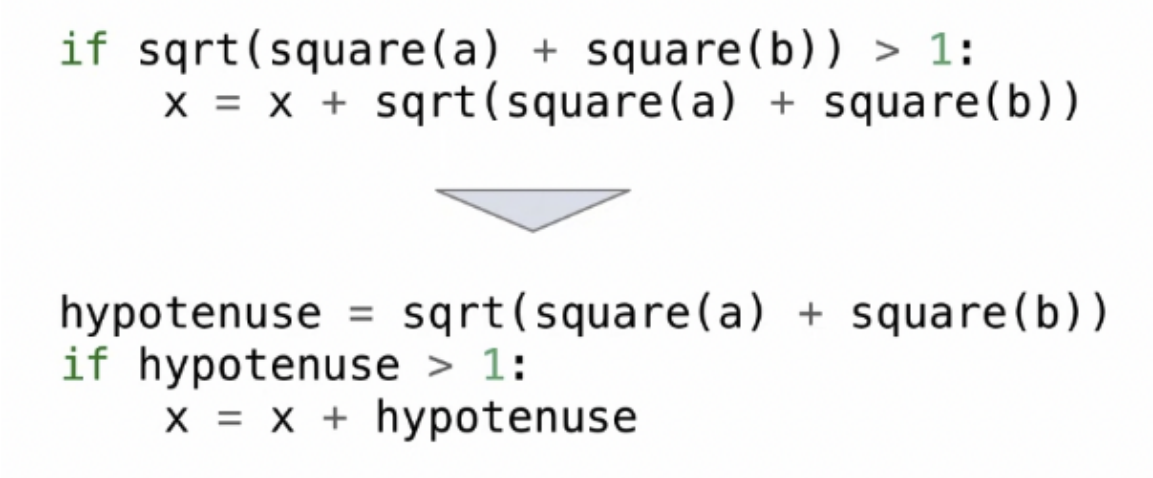
1. 参数的形式
2. 这个函数的行为/返回值

函数的命名： 什么才算是一个函数的好名字呢？

1. 名字应该能体现这个函数的功能
2. 函数的`docstring`中应该解释一下各个变量的含义

对于比较复杂的长式子，我们最好也给它起个名字

例如：



总而言之，要让人类能读懂你写的代码

Higher-Order Function Example: Sounds

CS61A mario歌曲名场面， 用于展示高阶函数

> https://www.youtube.com/watch?v=TC_JcE42R2s&list=PL6BsET-8jgYUC2Gl9J9Jo_JNLXWl7RcSs&index=3

Lecture 7: Function Examples

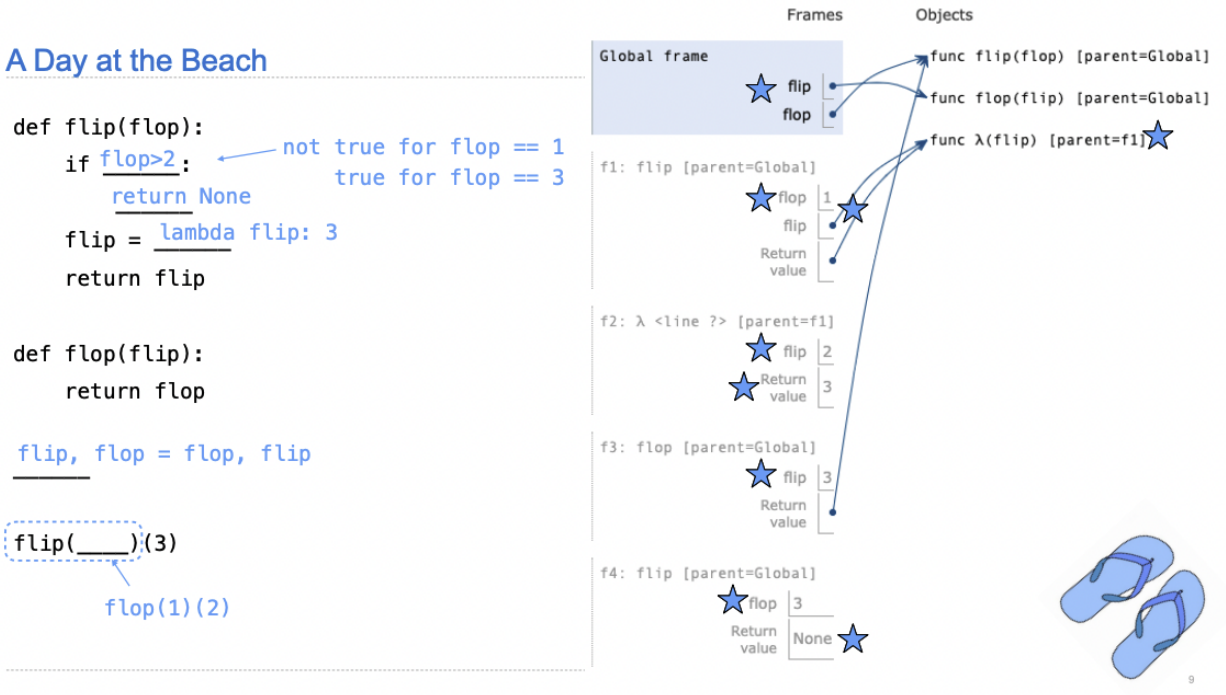
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c4592753-7e4a-468f-b9e9-7b37e98403ab/07-Function_Examples_1pp.pdf

Describing Functions

这里就是简单介绍了一下怎么用自然语言来描述一个函数的行为

Generating Environments Diagram

介绍了一个复杂的例子，如下图所示，你需要根据右边的Environments Diagram来还原左侧的代码



Implementing Functions

介绍了一下函数完形填空怎么分析

先自己实现一遍，然后把自己的实现进行微调，修改为题目要求的template

Decorators

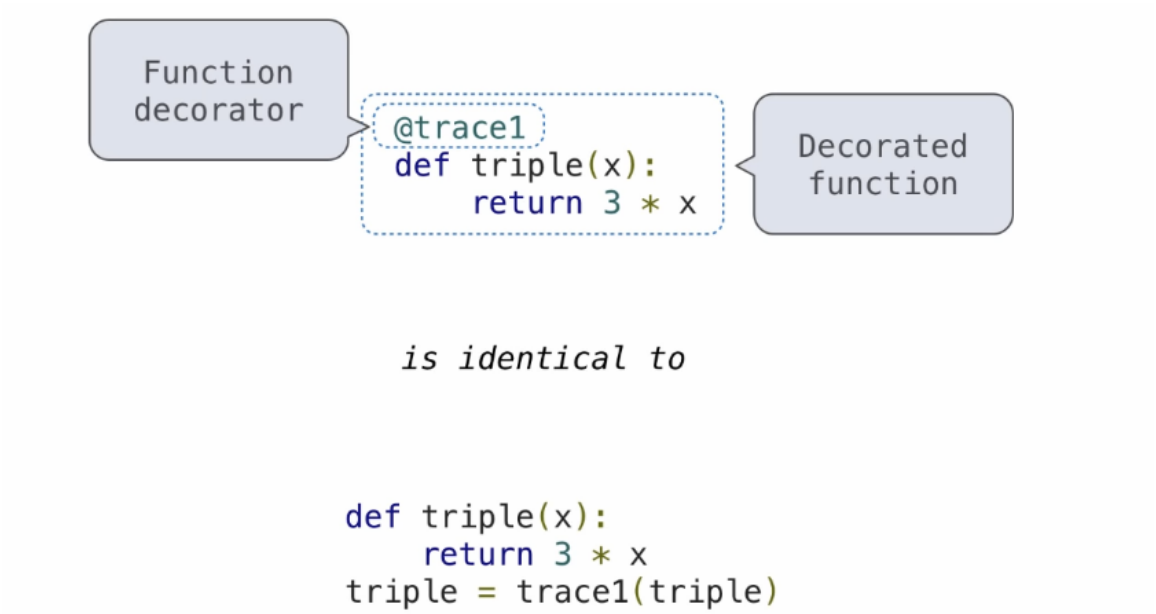
装饰器是Python对于高阶函数的利用，它可以化简调用高阶函数的代码

本质上，decorator就是一个返回函数的高阶函数。

- trace是一个接受函数为参数的高阶函数
- fn(x)是一个普通函数

那么

```
@trace              等  trace(fn)(3)
def fn(x):          价
    return ..       于

fn(3)               ==
```



这样使用高阶函数时就比较简单

例子如下：

```
1 def trace1(fn):
2     def traced(x):
3         print('Calling', fn, 'on Arguments', x)
4         return fn(x)
5     return traced
6
7 @trace1
8 def square_with_decorator(x):
```

```
  9     return x*x
 10
 11 def square(x):
 12     return x*x
 13



>>> trace1(square)(10)
Calling <function square at 0x101352f70> on Arguments 10
100
>>> square_with_decorator(10)
Calling <function square_with_decorator at 0x101352e50> on Arguments 10
100
>>>
```

Week 4

Lecture 8: Recursion

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4e1ddf40-ab03-48ed-8992-71be019a1fc0/08-Recursion_1pp.pdf

Recursive Functions

递归函数定义： 递归函数的函数体中 会调用它自身，无论是直接还是间接调用

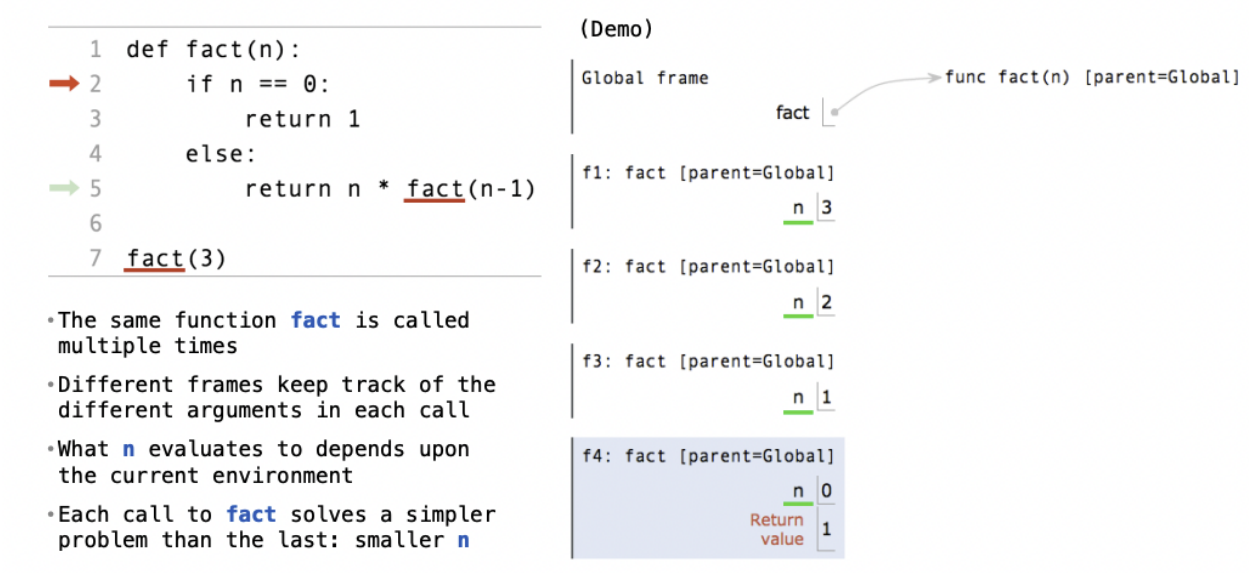
例子：

```python
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

递归函数的构成：

- 条件语句进行边界条件（base cases）的检查
- 边界条件是直接退出的，不会有递归调用
- 递归语句，这里调用函数本身

Recursion in Environment Diagrams



Verifying Recursive Functions

略过

Mutual Recursion

互递归： 指两个不同函数互相调用彼此

例子：Luhn Sum

在信用卡校验中使用了Luhn Algorithm. 从最右边的数字开始计算，如果它所在的位置是奇数，那么保持不变，如果是偶数，将其值乘2。这样可以得到一个新的序列，把这个新序列加起来再除与10就得到了校验位的值，这个值一定是10的倍数。如下所示

1	3	8	7	4	3	
2	3	1+6=7	7	8	3	= 30

代码：

```
def split(n):
    return n // 10, n % 10

def luhn_sum(n):
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return luhn_sum_double(all_but_last) + last

def luhn_sum_double(n):
    all_but_last, last = split(n)
    last = last * 2
    if last >= 10:
        last = last % 10 + last // 10
    if n < 10:
        return last
    else:
        return luhn_sum(all_but_last) + last
```

Recursion and Iteration

递归和循环的关系

循环是递归的特殊情况

Helper Function in Recursion

当我们需要递归求解，但函数的api已经定好的时候，我们可以进一步封装一下。定义一个helper函数帮助我们进行递归求解，外层的函数只需要return的时候调用一下helper函数即可。如下：

```
def shifty_shifts(start, goal, limit):
    """A diff function for autocorrect that determines how many letters
    in START need to be substituted to create GOAL, then adds the difference in
    their lengths.
    """

    def helper(start, goal, limit, num):
        if len(start) == 0:
            return num + len(goal)
        elif len(goal) == 0:
            return num + len(start)
        elif num > limit:
            return num
        else:
            if start[0] == goal[0]:
                return helper(start[1:], goal[1:], limit, num)
            else:
                return helper(start[1:], goal[1:], limit, num+1)

    return helper(start, goal, limit, 0)
```

Lecture 9: Tree Recursion

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6bef7596-7727-456d-99cc-a4c0ff645eb0/09-Tree_Recursion_1pp.pdf

Order of Recursive Calls

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

写递归函数的时候，把边界情况放到前面

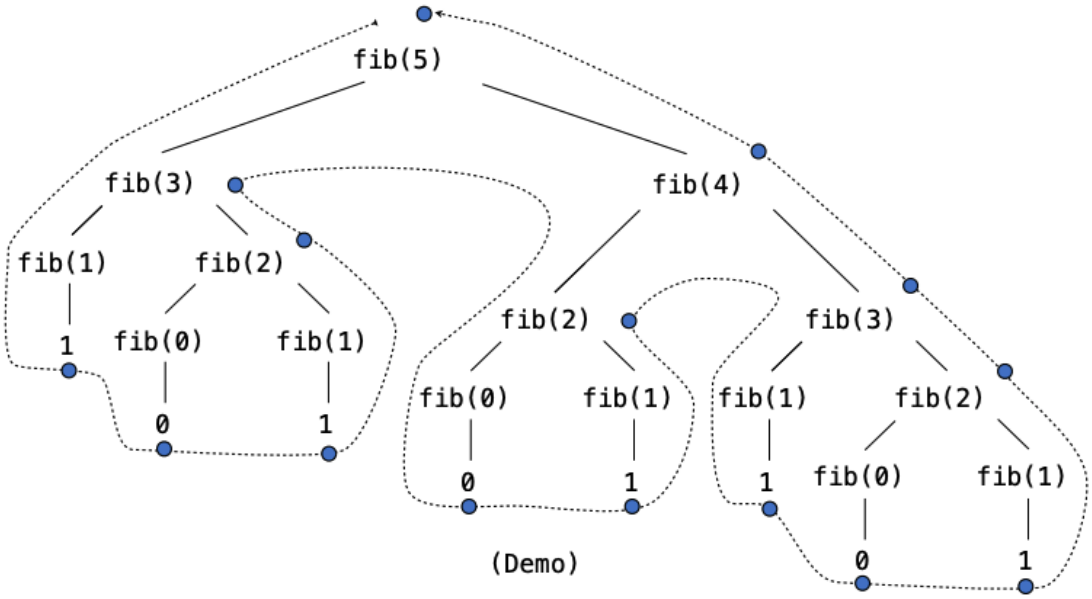
Example: Inverse Cascade

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow =   lambda n: f_then_g(grow,  print,  n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

Tree Recursion

树状的递归：递归的调用可以构成树这种数据结构，当递归时多次调用自己就会构成树递归

例如斐波那契数列就是树状递归：



Example: Counting Partitions

counting partitions 就是对于一个正整数n， 可以用小于等于m的正整数组合出n，这样的组合有几个的问题。(返回组合的个数)

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- 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
```

(Demo)

Week 5

Lecture 10: Containers

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1ee5843e-fc12-490e-92ff-aae4e2feda86/10-Containers_1pp.pdf

List

list是Python 的 built-in data type

角标从0开始，使用len(list)获得list长度

list的操作：

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

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

The number of elements

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

An element selected by its index

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

```
    >>> getitem(digits, 3)
    8
```

Concatenation and repetition

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

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

Nested lists

```
    >>> pairs = [[10, 20], [30, 40]]
    >>> pairs[1]
    [30, 40]
    >>> pairs[1][0]
    30
```

⚠️ Python中的sequence（序列）都可以切片，如list[a:b]代表list的[a, b）切片。但特别需要注意的是**空序列可以切片，但不能按角标取值，会出现index out of range**

Containers

list是一个container，可以contain其他的东西

你可以使用built-in operator `in` 来判断某个元素是否在container里

```
>>> digits = [3, 1, 8, 8]
>>> 1 in digits
True
>>> [1, 8] in digits #注意in只能作用于container里的元素
False
>>>
```

For Statements

for语句是Python引入来遍历序列的一种新语法

结构如下：

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

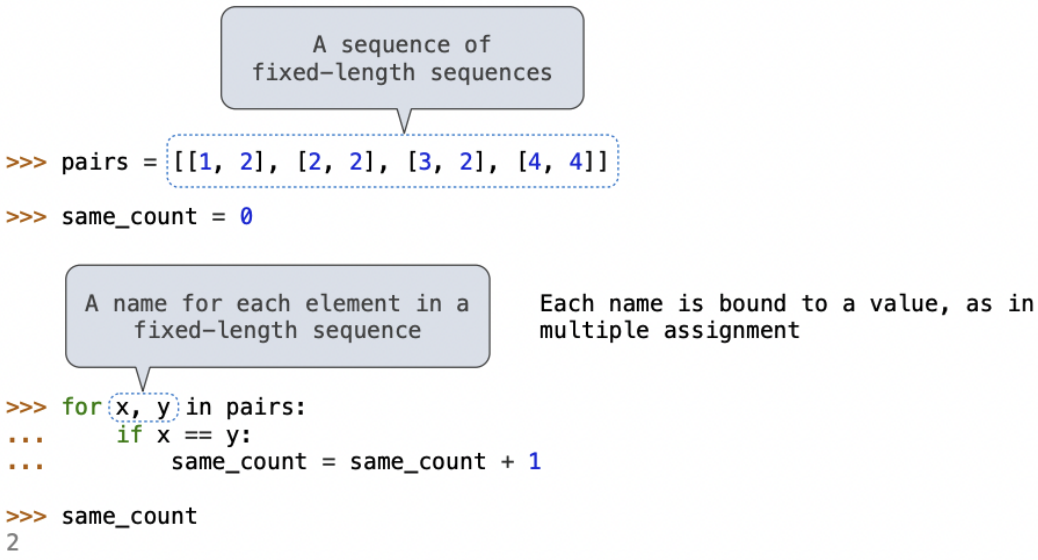
1. Evaluate the header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the current frame
   B. Execute the <suite>

例子：

```
 1 def count(s, value):
 2     count = 0
 3     for element in s:
 4         if element == value:
 5             count += 1
 6     return count
```

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 1, 1, 1]
>>> count(a, 1)
4
>>>
```

For循环中的序列解包



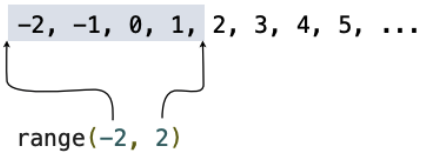
Range

Ranges是另一种序列类型（注意不是list，是单独的对象）

Range是用来表示连续整数序列的

用法：

A range is a sequence of consecutive integers.*



Length: ending value − starting value

(Demo)

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

⚠️ range(a, b)函数里的两个参数代表的整数序列是[a, b)
range(a)代表的整数序列是[0, a)

range还有一个比较特殊的用法,如下：

```
1 def cheer(n):
2     for _ in range(n):
3         print('Go Bears!')
4
```

```
(base) ➜  Week5 git:(master) ✗ python -i range.py
>>> cheer(3)
Go Bears!
Go Bears!
Go Bears!
>>>
```

Recursive Sums

Pass

List Comprehensions

列表表达式

例子：

```
>>> odds = [1, 3, 5, 7]
>>> [x+1 for x in odds]
[2, 4, 6, 8]
>>> [x+1 for x in odds if 25 % x == 0]
[2, 6]
>>>
```

Strings

string是对于字符数据的一种抽象

Python的源文件就是就是string这个数据类型组成的，如下：

```
>>> 'curry = lambda f : lambda x: lambda y: f(x, y)'
'curry = lambda f : lambda x: lambda y: f(x, y)'
>>> exec('curry = lambda f : lambda x: lambda y: f(x, y)')
>>> from operator import add
>>> curry(add)(3)(4)
7
>>>
```

有三种方式表示string

- 单引号： `'I am string'`
- 双引号： `" I am string"`

- 三个双引号： 可以表示多行，如下

```
>>> """ The Zen of Python
claims, Readability counts"""
```

用 `\` 来表示转移符， `\` 转义它后面跟着的字符，如 `\n` 转义了n， `\n` 这个整体代表换行

⚠️ String也是序列的一种，可以用 `in` ， `for` ， `len` 等关键词

习题： String Reversal

使用递归反转一个string

```
1 def reverse(s):
2     if len(s) == 1:
3         return s
4     else:
5         #print(s[1:])
6         return s[-1] + reverse(s[:-1])
7
```

Lecture 11: Data Abstraction

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/fc5861fb-d4af-4c5d-9413-7d7f0208330e/11-Data_Abstraction_1pp.pdf

Data Abstraction

数据抽象是一种抽象的方法，程序中的数据被隔离开来成为两个部分：

- 数据表示： How data are represented (as parts)
- 数据操作： How data are manipulated (as units)

总而言之：数据抽象就是一种抽象的方法，它在**数据的表示**和**数据的操作**之间建立抽象的隔离

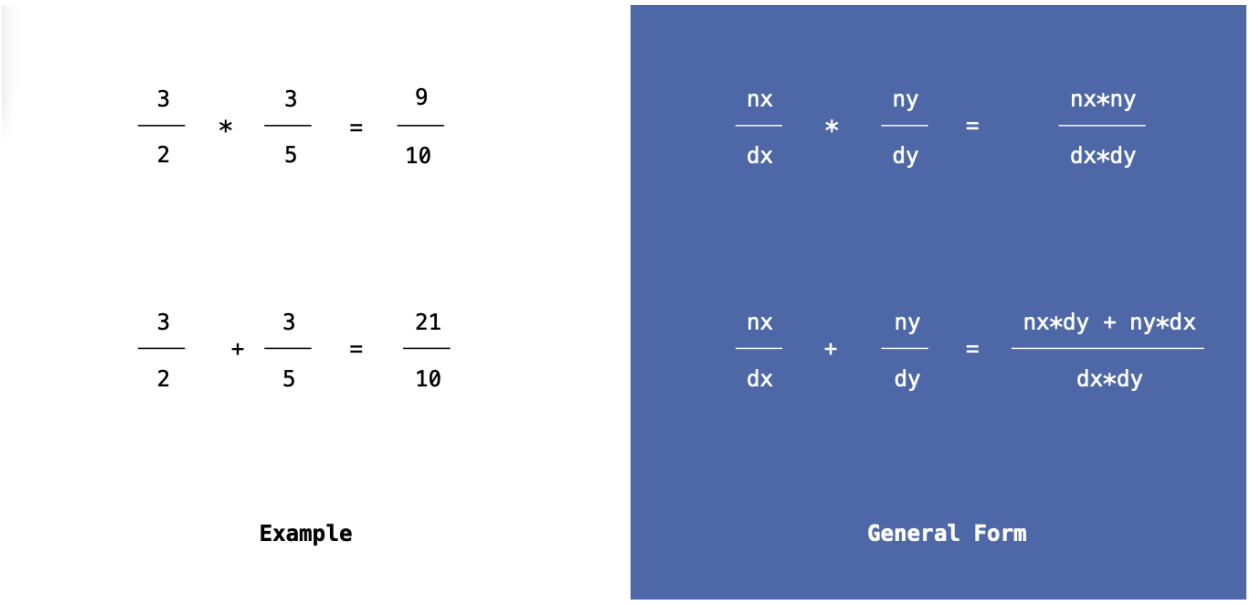
接下来我们举个数据抽象的例子：

接下来我们要分抽象层次去构建出有理数这个数据类型

1. 理解**有理数**

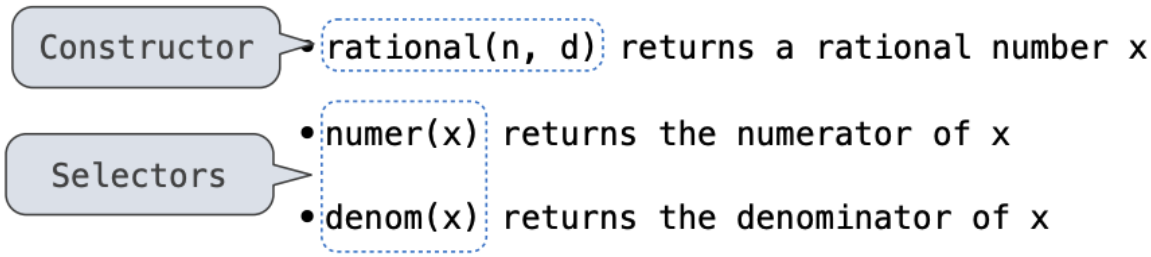
我们永远可以把有理数表示成分数的形式，如下

$$\frac{\text{numerator}}{\text{denominator}}$$



2．构建底层表示

为了表示有理数这个数据类型，我们需要Constructor 和 selector这两种函数



```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Select item from a list

3．构建有理数的操作子

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)

def print_rational(x):
    print(numer(x), '/', denom(x))

def rationals_are_equal(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

These functions implement an abstract representation for rational numbers

Pairs

pair 就是把两个东西组合成一个整体（unit）

如：`pair = [1, 2]`

Unpacking list

`x, y = pair`

getitem

`getitem(pair, 0)`

为了解决上面的那个有理数的例子，我们还需要引入分数化简，注意我们只需要修改有理数的 `Constructor` 函数即可

Abstraction Barries

抽象的层次很重要

我们可以把有理数抽象成以下这几个层次

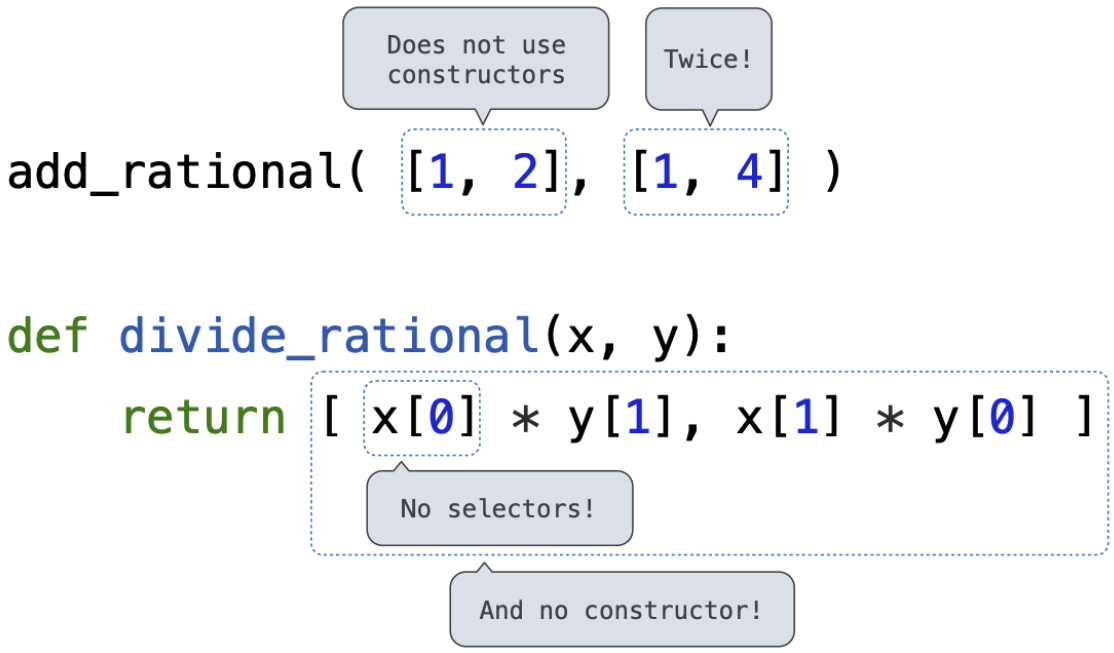
暴露给用户的功能　　用户需要知道的数据形式　　用户可以使用的函数，方法

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

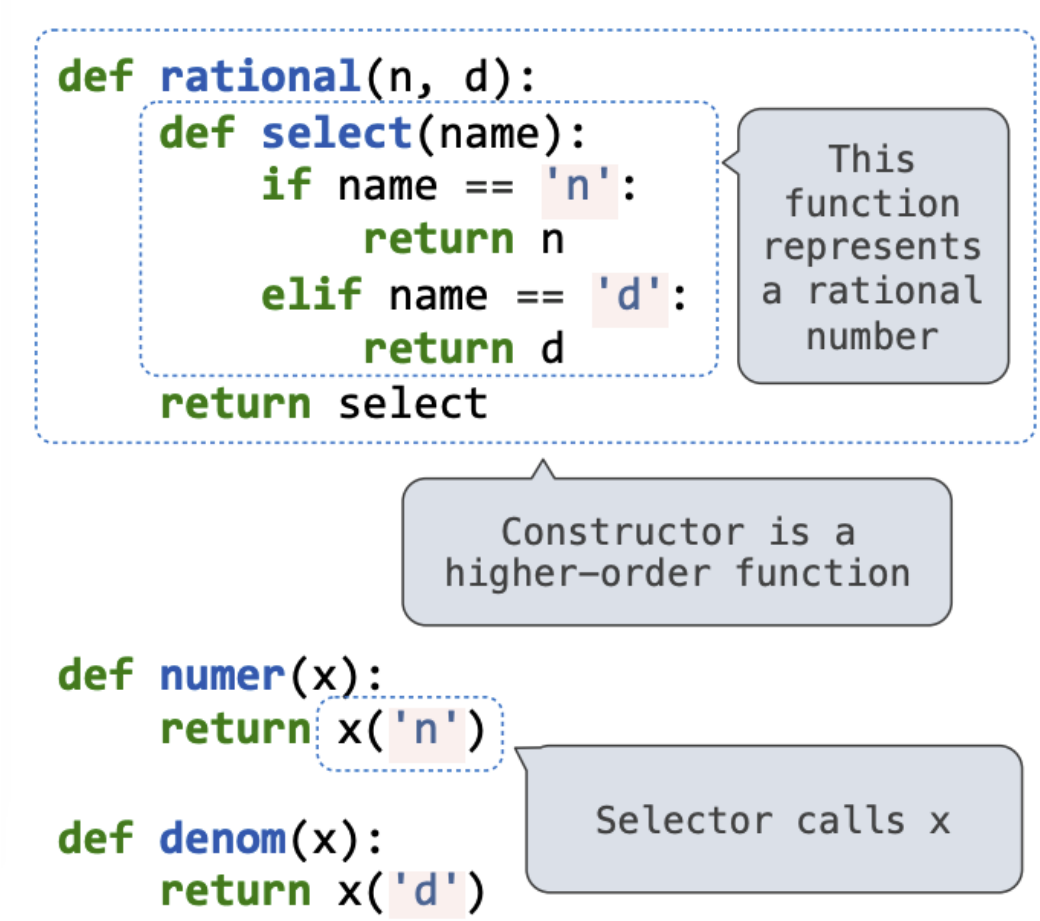
破坏抽象层次非常不好，这样打破了隔离，上层用户不应该了解到下层的具体实现

举个破坏抽象层次的例子：



Data Representation

如下图，我们可以把使用list实现的rational改为使用函数实现，而且这样做是不会影响到其他抽象层的



Dictionary

字典就是一种map映射

⚠️ 字典默认是无序的

字典的方法：

- 字典可以使用 `dic[key]` 的形式来进行索引，得到value
- 查看key: 通过`dic.keys()` 来查看字典中的keys
- 查看key-value pairs: 通过 `dic.items()` 来查看字典中的pairs
- 查看某个key是否在字典中： 使用 `key in dic`

字典推导式，例如：

```
>>> {x: x*x for x in range(8)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
>>>
```

字典的限制：

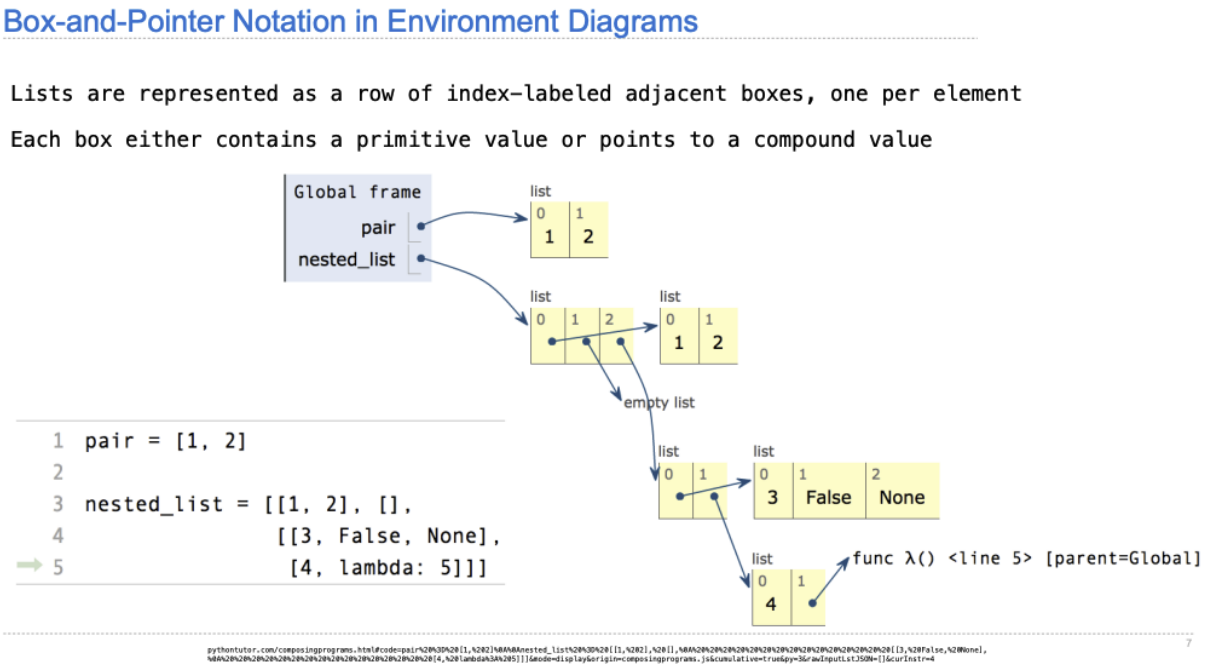
- 字典的key不能是 **可变类型**
- 字典里的 **key是唯一的**

Lecture 12： Trees

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e2a495d3-ea0e-4689-bde9-569cb755fb90/12-Trees_1pp.pdf

Box-and-Pointer Notation

由于Python有闭包性质（list里可以包含list），为了追踪list里的内容，我们需要扩展Week2中提到的environment diagram。如下图所示



Slicing

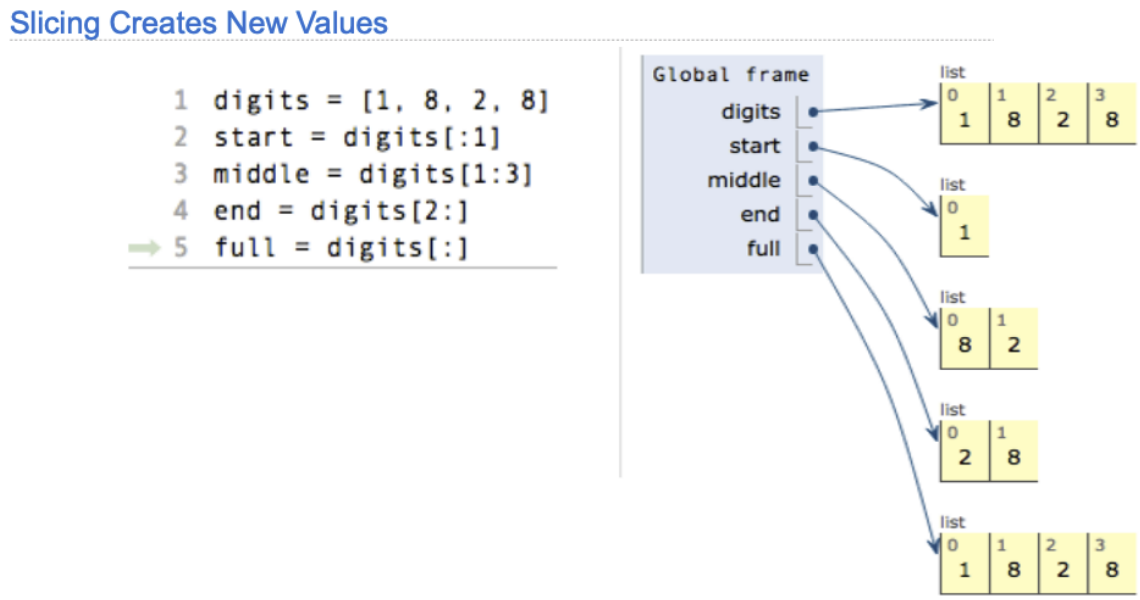
切片是Python中对于 `Sequence` 的一种操作

例如：

```
>>> odds = [1, 3, 5, 7, 9]
>>> odds[1:3]
[3, 5]
>>> odds[1:]
```

```
[3, 5, 7, 9]
>>> odds[:]
[1, 3, 5, 7, 9]
>>>
```

⚠️ 切片永远会创造新的值，如下图



Processing Container Values

这里介绍一些可以方便处理容器的内置函数

- `sum(iterable[, start]) -> value`

  返回`start + iterable`里所有的值，`start`默认是`0`

  需要注意以下这种情况：

```
>>> sum([[1, 2], [3, 4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> sum([[1, 2], [3, 4]], [])
[1, 2, 3, 4]
>>>
```

- max 函数有两种形式

  `max(iterable[, key==func]) -> value`

  `max(a, b, c,...[, key=func]) -> value`

  这里的key函数是指，算max的时候把每个值都apply一遍这个func，以此为标准，返回出apply过这个func的那个最大值。如下：

```
>>> test = [-4, -2, 0, 1, 2]
>>> max(test, key= lambda x: x*x)
-4
>>>
```

- `all(iterable) -> bool`

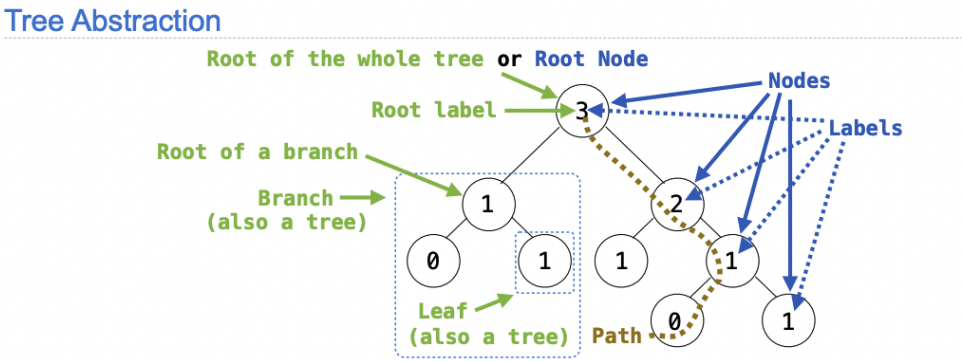
  如果对于iterable里的每一个元素x, bool(x) == True， 那么返回True

  如果iterable是空的，你们也是True

```
>>> [x < 5 for x in range(5)]
[True, True, True, True, True]
>>> all([x < 5 for x in range(5)])
True
>>>
```

Trees

树的定义：



Recursive description (wooden trees):

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**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

树的实现：

Implementing the Tree Abstraction

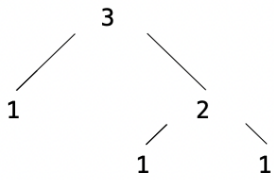
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list

- A **tree** has a root **label** and a list of **branches**
- Each branch is a tree




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

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

(Demo)

15

树的处理：

Pass

Example: Printing Trees

Pass

Example: Summing Paths

⚠️ 递归函数的return的值也可以添加一个参数，每次递归调用函数的时候更新那个参数

Week 6

Lecture 13: Binary Numbers(Optional)

这节是讲计算机数据的存储形式的（二进制）

Pass

Lecture 14: Circuits(Optional)

这节是讲电路的

Pass

Lecture 15: Mutable Values

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/03e46767-0bf7-4f21-9d65-badc73eaf5ad/15-Mutable_Values_1pp.pdf

Objects

Python中的一切都是对象

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes
- A type of object is called a class; **classes** are first-class values in Python
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object
  - All **objects** have **attributes**
  - A lot of data manipulation happens through object **methods**
  - Functions do one thing; objects do many related things

Example: String

字符串是对象

```
>>> s = 'Hello!'
>>> s.upper()
'HELLO!'
>>>
```

Mutation Operations

有些对象是可变的

对象只存在一份。如果它发生了改变，那么所有被绑定到这个对象的名字也会被影响到，如下：

```
>>> home = ['beijing', 'dongcheng', '166']
>>> old_home = home
>>> home.pop()
'166'
>>> old_home
['beijing', 'dongcheng']
>>>
```

如果一个对象发生了改变，那么我们就称之为发生了 ”mutation“

函数也可以改变对象，发生mutation

```
>>> four = [1, 2, 3, 4]
>>> def mystery(s):
...     s.pop()
...     s.pop()
...
>>> len(four) #four就是一个object
4
>>> mystery(four)
>>> len(four)
2
>>>
```

Tuples

元组也是序列，但是他们是 **不可变** 序列

不可变是指其中的元素是不能修改的

看个例子：

```
>>> (3, 4, 5, 6)
(3, 4, 5, 6)
```

```
>>>
```

事实上在Python中任何的被逗号 `,` 隔开的都是被视为元组的

因为tuple也是sequence，所以 `+` `in` `:` 等操作都是可以使用的

如果不可变类型的序列的元素是可变类型的，那么不可变类型的序列就可以更改里面的元素了

```
>>> s = ([1, 2], 3)
>>> s[0] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s[0][0] = 4
>>> s
([4, 2], 3)
>>>
```

Mutation

在Python中到底什么才是一样的，什么才是不一样的呢？

如下面这个例子，左边的是一个东西；右边的尽管内容一样，但它们也不是一个东西

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
True
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

Identity Operators

你如何来判断两个东西到底是不是一个呢？

引入新的 `is` 操作符

Identity

<exp0> is <exp1>

evaluates to True if both <exp0> and <exp1> evaluate to the same object

Equality

<exp0> == <exp1>

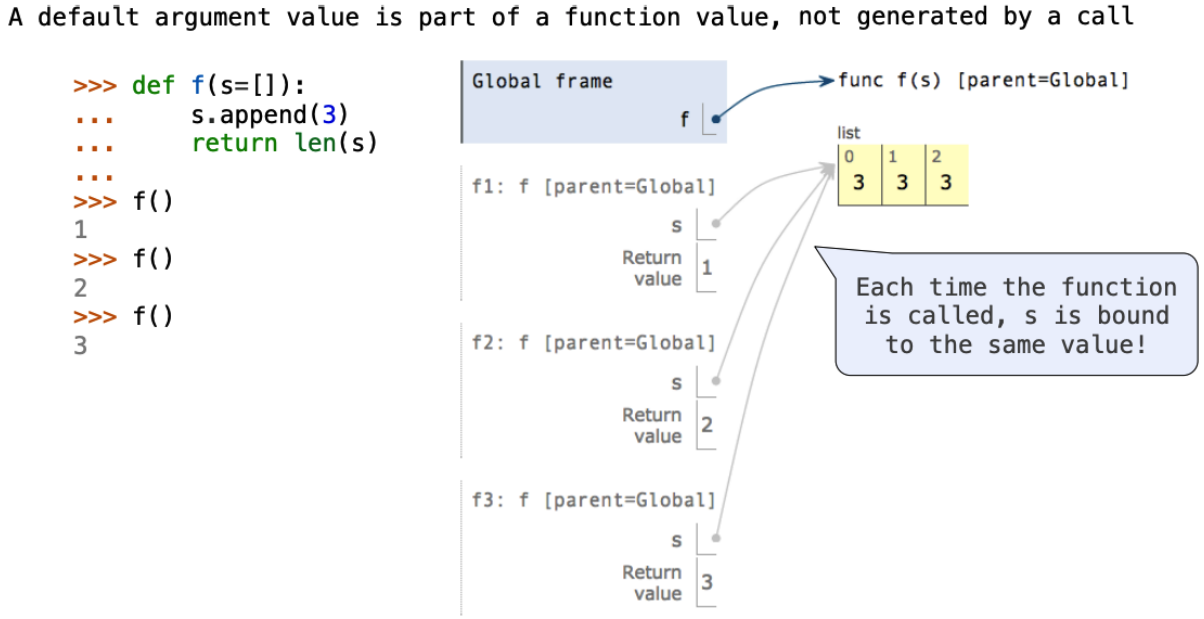
evaluates to True if both <exp0> and <exp1> evaluate to equal values

Identical objects are always equal values

注意这里的 `is` 并不等于 `==`

Mutable Default Arguments are Dangerous

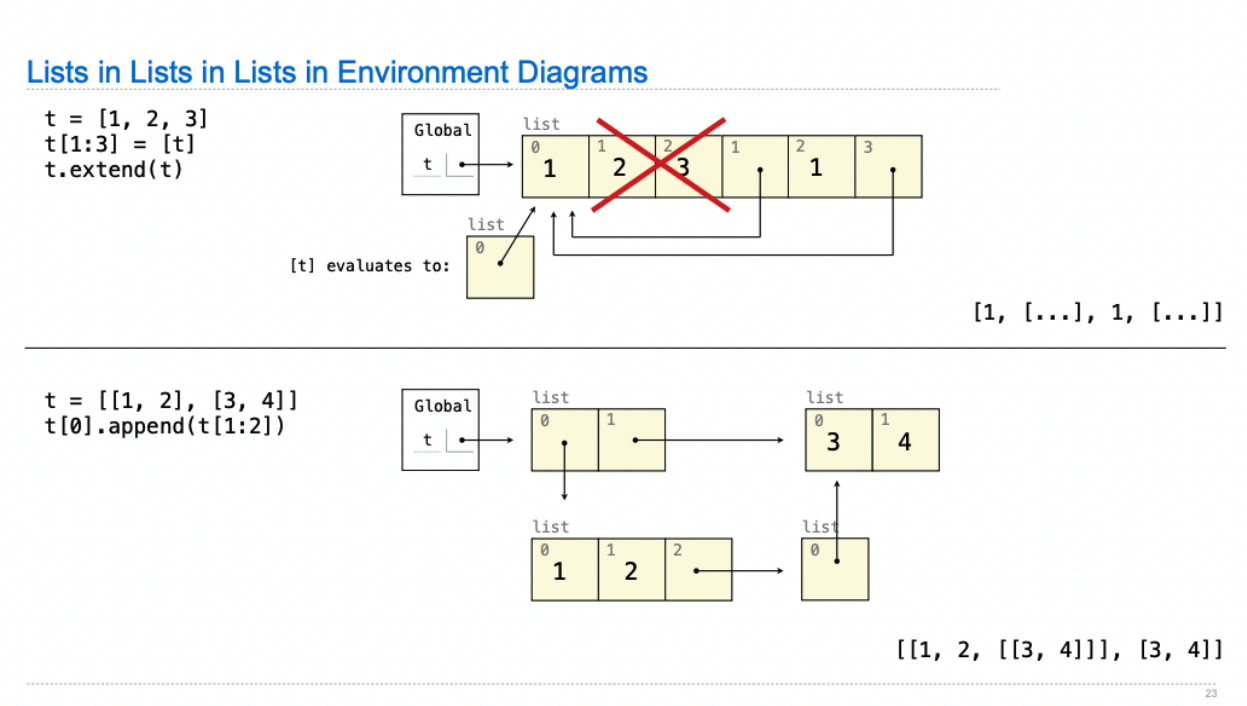
函数的参数如果有可变对象作为默认值的话，在函数体的内部尽量不要去修改这个默认值，这样你每次调用的时候结果都会不一样。如下图：



函数的默认参数是这个函数内部值的一部分，并不会随每次函数的调用而产生一份新的。这也就是为什么上面的例子里每次调用 `f()` 的结果都不一样

Example： Lists Mutations

两个比较复杂的例子：



Week 7

Lecture 16: Mutable Functions

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/36ece7d9-3e77-4965-aef6-a17f4651bb38/16-Mutable_Functions_1pp.pdf

A Function with Behavior That Varies Over Time

我们如何让函数的功能动态变化呢？需要`nonlocal` 变量

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

Re-bind balance in the first non-local frame in which it was bound previously

声明nonlocal，就是声明这个变量不在当前的frame里，而是往上（parent frames）迭代直到找到这个对应的变量

nonlocal的语法：

```
nonlocal <name>, <name>, ...
```

效果：此语句之后的赋值语句（=）改变的是第一个非本地frame中同名的变量

要求：

1. 声明的非本地变量必须在之前定义过
2. 本地的变量名和非本地的变量名不能重合，发生碰撞

赋值语句的几种情况：

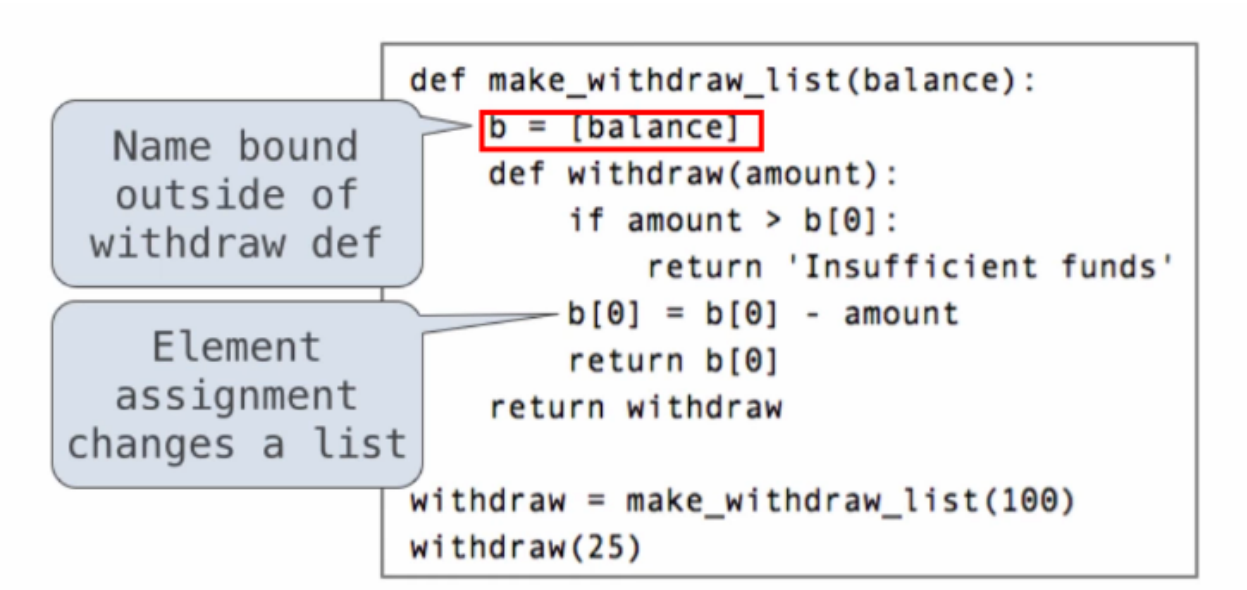
The Many Meanings of Assignment Statements

`x = 2`

Status	Effect
•No nonlocal statement •"x" **is not** bound locally	Create a new binding from name "x" to object 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

10

可变数据类型即使不使用nonlocal声明也可以被改变：



前面的必须要使用nonlocal是因为数字是不可变类型

Referential Transparency

引用透明性：如果一个表达式在被其返回值代替，而程序的行为，结果没有发生变化，那么我们就叫这种性质为引用透明性。这需要这个表达式是对于同一个输入永远有同样的输入，而且还不能有side effects泄露出来。

如下图中的代码，b(3) 就不能被10取代，因为b()由于nonlocal变量的存在，它丧失了引用透明性。

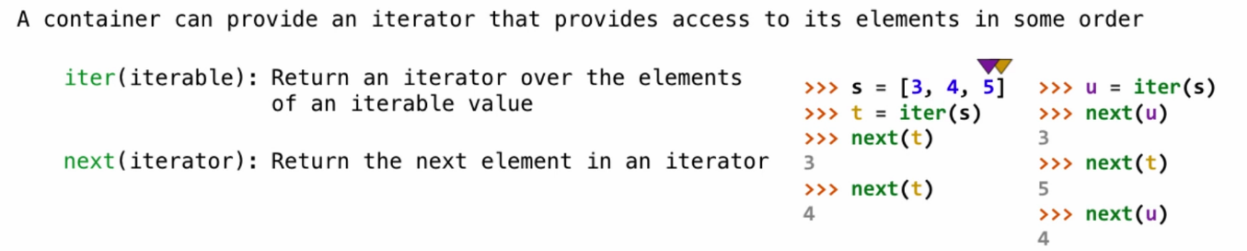
```
def f(x):
    x = 4
    def g(y):
        def h(z):
            nonlocal x
            x = x + 1
            return x + y + z
        return h
    return g

a = f(1)
b = a(2)
total = b(3) + b(4)
```

Lecture 17: Iterators

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/23281c1e-4d8d-408c-9a54-b288434d4daa/17-Iterators_1pp.pdf

Iterators

序列数据（sequence data）都可以用迭代器（iterator）来表示



使用next()访问到最后会触发StopIteration
迭代器不会记录之前访问过的元素，可以认为迭代器会丢弃访问过的元素

Dictionary Iterations

可迭代量（iterable value）：任何可以被传递到 `iter()` 从而得到迭代器的量

迭代器（iterator）：任何可以被传递到 `next()` 中的东西；所有的迭代器都是可变的

对于字典来说，它的keys，values，items 都是可迭代量

值得注意的是：

- 老版本的python中的字典是无序的（Python 3.5 and ealier）
- 现在的Python中的字典是有序的，意思是说最后添加的key-value会被最后迭代到（Python 3.6+）

- 由于字典内部实现的结构，当改变字典的size的时候，字典产生的迭代器将抛出异常，无法工作

For Statements

for循环具体做的工作就是不断的使用next()方法，直到遇到StopIteration

使用for来遍历 **可迭代量** 的时候都会重新生成一个可迭代器

例子如下：

```
>>> lst = [1, 2, 3]
>>> for i in lst:
...     print(i)
...
1
2
3
>>> it = iter(lst)
>>> next(it)
1
>>> for i in it:
...     print(i)
...
2
3
>>>
```

Built-In Iterator Functions

```
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(first_iter, second_iter):    Iterate over co-indexed (x, y) pairs
                  reversed(sequence):      Iterate over x in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container

                      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
```

Lazy Computation： 只有真正用到这个值（迭代到）时才会具体计算

lazy computation是很方便的，因为你可以定义无穷多的数据来进行计算，由于其不是立刻计算的，而是一个一个计算的，可以叫停，所以不需要无穷多的时间来进行计算

Generators

A generator is a special kind of iterator.

generator看起来就像一个函数，只不过函数使用 `return` 而generator使用 `yield`

当调用**生成器函数**的时候会返回一个生成器，这个生成器不会去执行函数体内的内容。对这个生成器采用 `next()` ，这个生成器会遍历它的 `yield`s

但是当 `yield` 退出函数之后，我们还会记录下来这个frame的环境，当你再调用 `next()` 时这个生成器会从上次执行退出的位置继续执行（也就是 `yield` 之后的第一条语句）

如下图中的代码 `even += 2` 就是第二个 `next()` 后第一个执行的语句

```
>>> def evens(start, end):
...     even = start + (start % 2)
...     while even < end:
...             yield even
...             even += 2
...
>>> t = evens(2, 10)
>>> next(t)
2
>>> next(t)
4
>>> next(t)
6
>>>
```

注意：不可以直接 `next(generator())` 因为这样这个生成器是没名字的，下次再调用只会返回一个新的生成器，我们必须 `g = generator() next(g)` 才可以

Generators & Iterator

`yield from <iter>`

`yield from` 后面需要加的是可迭代对象，它可以是普通的可迭代对象，也可以是迭代器，甚至是生成器。

`yield from` 后面加上可迭代对象，他可以把可迭代对象里的每个元素一个一个的yield出来，对比yield来说代码更加简洁，结构更加清晰。

例子：

```
>>> def prefix(w):
...     if w:
...         yield from prefix(w[:-1])
...         yield w
...
```

```
>>> list(prefix('dogs'))
['d', 'do', 'dog', 'dogs']
>>>
```

Lecture 18: Objects

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/f7262bd2-5ca0-4357-843d-5a2f0e84877e/18-Objects_1pp.pdf

Object-Oriented Programming

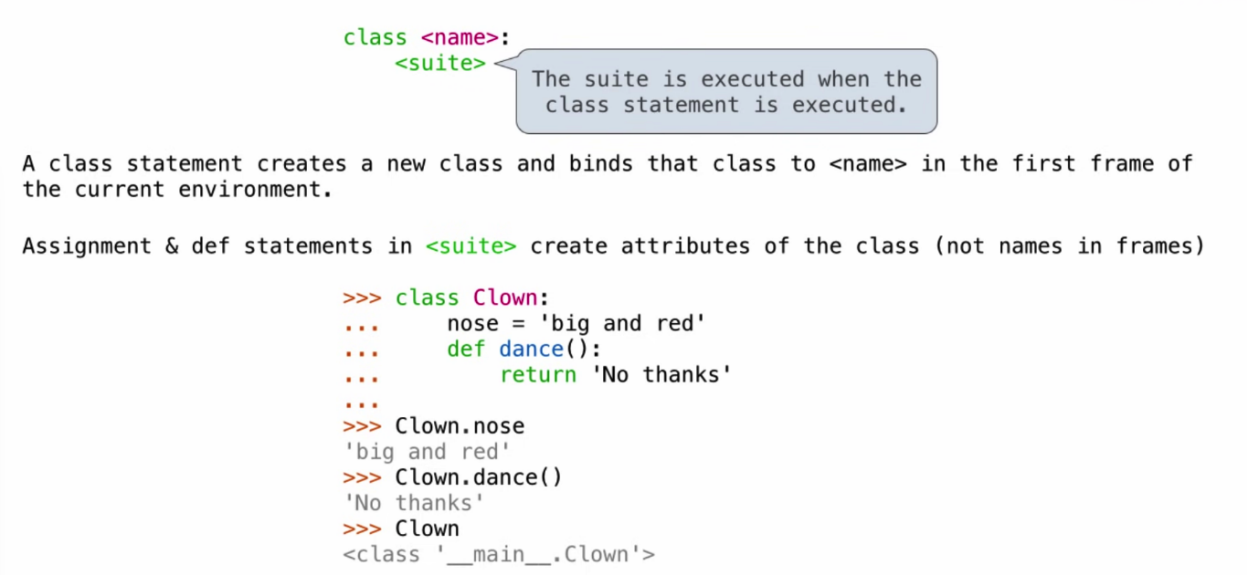
面向对象编程是一种组织程序的方法（还有其他的组织形式，如函数式编程），这需要：

- 建立数据抽象
- 把信息以及其相关的操作放到一起

Class

类是为了给实例提供模板的

Class Statements



class <name>的第一个字母需要大写

当一个class被调用（也就是call了这个class，如 `Clown()` ）之后：

1. 首先一个这个类的实例被创建了
2. 之后这个类的 `__init__` 方法会被自动调用来初始化这个实例，这个方法中的第一个参数是固定的 `self` ，其自动代指1中创建的那个实例， `self` 之后可以跟随着其他初始化所需要的参数

如下图：

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

Object Identity

用户创建的所有类的实例都是有唯一的身份的；如果 `a` 是一个类的实例， 那么如果你binding， 即 `c = a` ，那么实例还是只有一个，只是现在多了一个名字 `c` ，也就是说 `a is c` 为 `True`

Methods

方法就像函数，但是其是在被定义在类里的，如 `__init__` 就是一个方法

如下图：

```
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
```

当你使用 `.` 来调用类的方法的时候， `self` 这个参数是会自动传递过去的，不用手动传，传过去的参数其实就是你当前的这个实例；如果不用 `.` 的话，就要用这种方式来表达 `Account.deposit(john, 100)`

Dot Expression

Objects receive messages via dot notation.

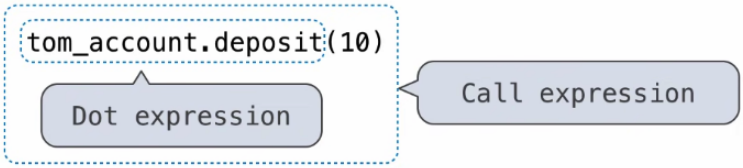
Dot notation accesses attributes of the instance **or** its class.

<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>.



Attributes

属性其实就是被类或者实例存储起来的数据

Accessing Attributes

Using getattr, we can look up an attribute using a string

```
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

getattr and dot expressions look up a name in the same way

`getattr` 等价于 `.`

当你访问一个属性的时候可能返回：

- 实例的一个属性
- 类的一个属性
- 会先去实例里找，再去类里找

Methods and Functions

Python会区分：

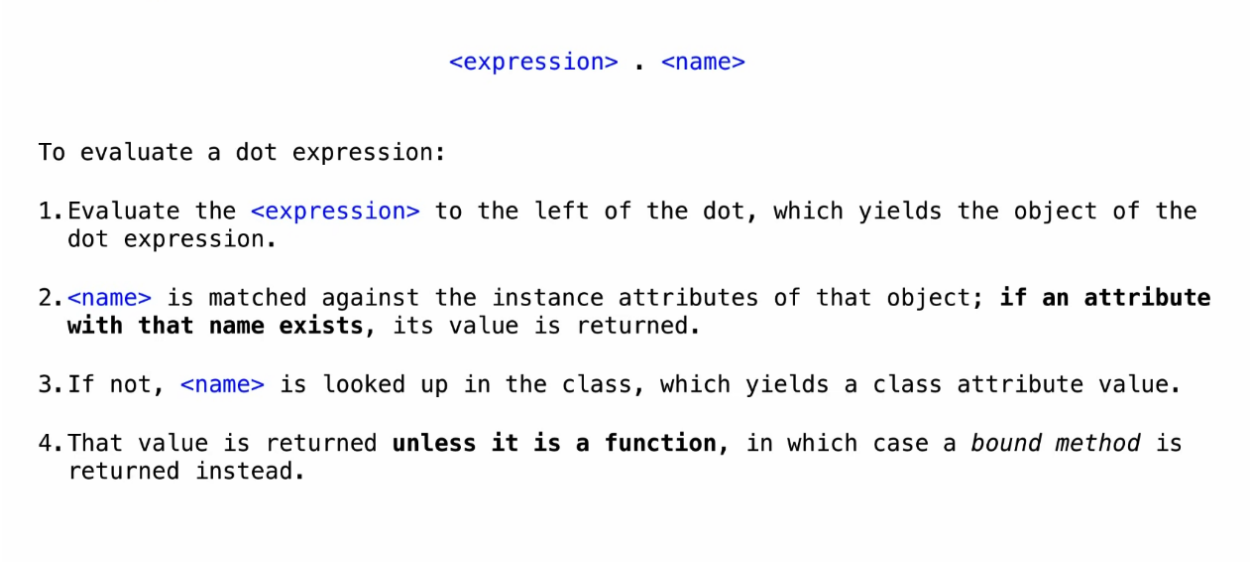
- 函数
- Bound methods: Object + Function

例如：

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


>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1000)
2011
```

Looking Up Attributes by Name

我们这里介绍一下Accessing Attributes里的**looking up**



如上图，其实就是先在实例里找，再在类里找

类里的属性是全实例共享的，它并不会复制到每个实例中，所以当你改变类的属性的时候，所有实例的这个属性都会被改变

```
>>> class Car:
...     color = 'Blue'
...     def __init__(self, size):
...             self.size = size
...
>>> a = Car(100)
>>> b = Car(200)
>>> a.color
'Blue'
>>> b.color
'Blue'
>>> Car.color = 'Red'
>>> a.color
'Red'
>>> b.color
'Red'
>>>
```

Week 8

Lecture 19: Inheritance

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2493774c-6fb6-4bf0-b70e-6862726f8313/19-Inheritance_1pp.pdf

Attributes

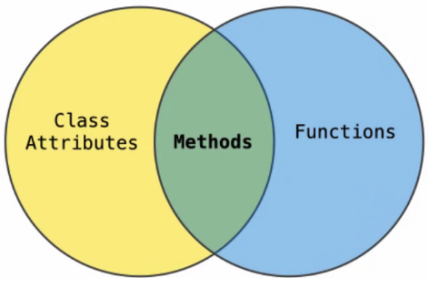
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

Terminology:



Python object system:

Functions are objects.

Bound methods are also objects: a function that has its first parameter "self" already bound to an instance.

Dot expressions evaluate to bound methods for class attributes that are functions.

注意：虽然定义上method也叫attributes， 不过我们习惯上还是把两者分开的，称为类的方法和类的属性

Looking Up For Attributes

```
<expression> . <name>

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.
```

Attributes Assignment

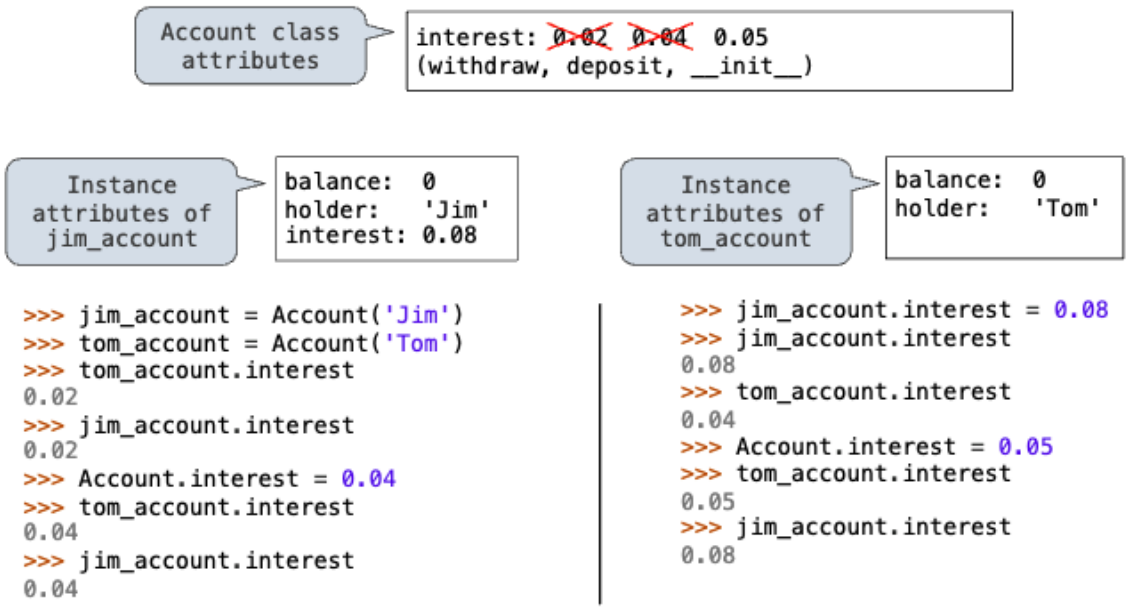
当给属性使用 `=` 赋值的时候，根据 `.` 左边对象不同分为以下这些情况：

- 如果是实例的话，那么改变实例的属性
- 如果是类的话，那么改变类的属性

注意：类的属性（定义在class `__init__` 之外的那些属性），在实例被创建出来的时候是不会被复制一份放到自己实例的frame里的，而是只有当你访问这个属性的时候再去class里找。 **但是如果你改变了实例的类属性，那么这个实例就会在自己的frame里创建一个新的属于自己的属性**

例子如下：

exp1:



exp2:

```
class Persion(object):
    count = 0

    def __init__(self):
        self.name = '人'


if __name__ == '__main__':
    p = Persion()
    p.count += 1

    print(p.count)
    print(Persion.count)

# output
1
0
```

Inheritance

继承是关联类的一种方法

继承的一般形式：

```
class <name>(<base class>):
    <suite>
```

我们说 class继承了base class

- 概念上来说，新的的子类共享了父类(<base class>)的属性
- 子类可以重写，覆盖某些特定的，继承自父类的属性
- 使用继承这个机制，我们可以实现**一个改写了父类特定的部分内容的子类**
- 当你call子类来创建实例的时候，如果子类没有 `__init__()` 方法，那么会调用父类的 `__init__()` 方法

如下面这个例子：

A CheckingAccount is a specialized type of Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class Account

```
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)
```

💡 **使用 `super()` 就相当于略过本层的同名方法，让Python帮你自动找到上一层的同名方法，而且self这个参数也会自动帮你传递过去（也就是不用再传self）**

💡 注意：子类并没有复制出父类所有的属性，它只是通过 **Looking up attributes names** 这个机制来调用那些子类和父类相重合的属性

举个例子：

```
>>> 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
```

Object-Oriented Design

Designing for Inheritance

- 不要重复，使用已经实现过的对象
- 被改写的方法也是可以通过 `<base class>.xxx` 的方法
- 注意，实例可以随时使用 `looking up` 机制来得到不同层次的属性

如下：

```
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)
```

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for specialized accounts

Inheritance and Composition

那么我们如何去分析什么时候重新写一个类（composition）；什么时候继承一个类去重构（Inheritance）呢？

- Inheritance is best for representing *is-a* relationships.
  - E.g., checking account **is a** specific type of account.
  - 所以要用继承
- Composotion is best for representing *has-a* relationships
  - E.g., a bank **has a** collection of bank accounts it manages.
  - 所以要用构成一个类

Attributes Lookup Practice

some exercise, see in https://www.youtube.com/watch?v=qFvC4SwbG4A&list=PL6BsET-8jgYXpV7vl4Pvo25wh0FKRlecx&index=6

Pass

Multiple Inheritance

在Python中class是可以继承多个父类的

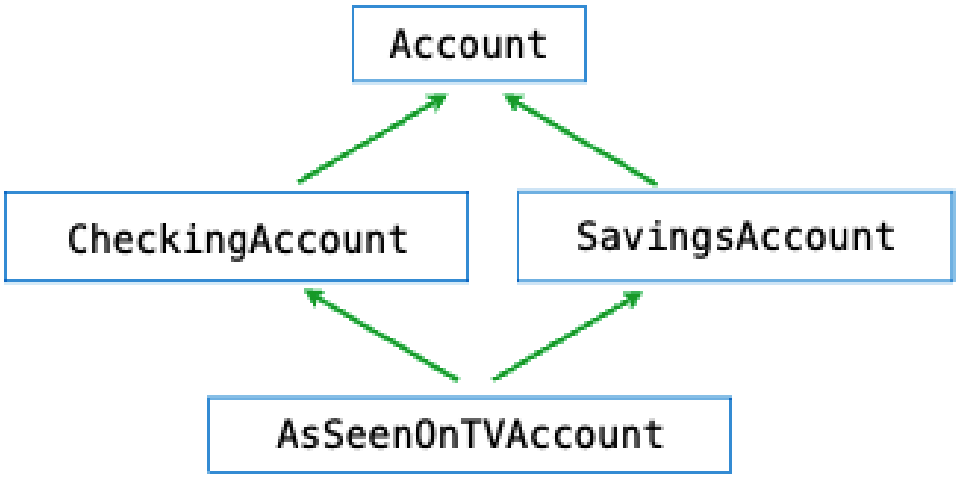
```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1                # A free dollar!
```

Instance attribute

SavingsAccount method

CheckingAccount method

```
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

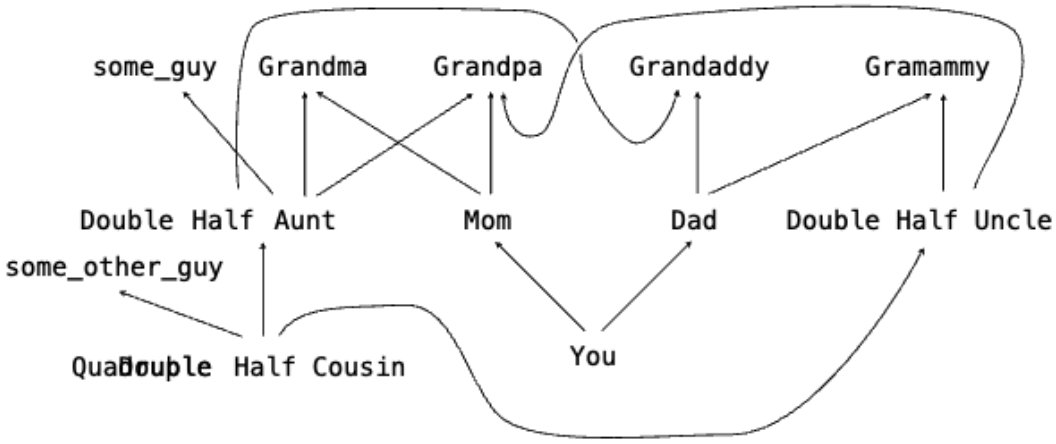


可以看到类的继承是可以得到这种”钻石”形状的图的，那么Python是如何解析的呢？

> 💡 Python继承的顺序为：从左到右，从上到下

在本题中继承的解析顺序为： `AsSeenOnTVAccount -> CheckingAccount -> SavingsAccount -> Account`

Complicated Inheritance



Moral of the story: Inheritance can be complicated, so don't overuse it!

Lecture 20: Representation

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/fc719055-1f8c-46d9-93a4-a5f5be455cc6/20-Representation_1pp.pdf

String Representations

字符串是很重要的，它表示了语言和程序

在Python中，所有的对象都有两种字符串的表达形式

- `str` 是面向人类的

- `repr` 是面向Python解释器的

大部分情况下 `str` 和 `repr` 都是一样的，但是也有例外，下面我们介绍一下

The repr String for an Object

`repr` 函数把对象转换为供解释器读取的格式

语法：`repr(objects) -> objets`

`eval()` 和 `repr()` 互为反函数

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

Some objects do not have a simple Python-readable string

```
>>> repr(min)
'<built-in function min>'
```

The str String for an Object

`str` 函数将对象转化为可供人类理解的字符串格式

语法：`str(objects) -> string`

实际上Python `print()` 就是调用 `str()` 函数

Human interpretable strings are useful as well:

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

The result of calling `str` on the value of an expression is what Python prints using the `print` function:

```
>>> print(half)
1/2
```

Polymorphic Functions

多态函数：一个可以接受许多不同形式数据作为参数的函数

上文中的 `str` 和 `repr` 都是多态函数，因为它们可以接受所有的objects

实际上，`str()` 和 `repr()` 只是调用了每个不同数据类型的同名方法 `__str__()` 和 `__repr__()` ，所以只要每个类都重写这两个特殊的方法，就可以实现多态了。

`repr` invokes a zero-argument method `__repr__` on its argument

```
>>> half.__repr__()
'Fraction(1, 2)'
```

`str` invokes a zero-argument method `__str__` on its argument

```
>>> half.__str__()
'1/2'
```

`str` 和 `repr` 的底层实现：

```
def repr(x):
    return type(x).__repr__(x)

def str(x):
    t = type(x)
    if hasattr(t, '__str__'):
        return t.__str__(x)
    else:
        return repr(x)
~
```

实际上，调用 `str` 和 `repr` 的时候会**忽略实例的同名方法**，而是利用 `type(x)` 的方法找到class的 `__str__` 和 `__repr__` 方法

Interfaces

接口是一组被分享的信息，定义好了这些信息的含义以及如何使用他们

比如说：`__str__` 和 `__repr__` 就向处理字符串提供了接口。

Special Method Names

Python中有些函数是类似于 `__<names>__` 这种有下划线的形式的，这是因为他们有特殊的，特定的 built-in 行为

如下图：

```
__init__        Method invoked automatically when an object is constructed
__repr__        Method invoked to display an object as a Python expression
__add__         Method invoked to add one object to another
__bool__        Method invoked to convert an object to True or False
__float__       Method invoked to convert an object to a float (real number)
```

实际上Python的优良扩展性也是来自于此，我们只需要重载用户定义的类里的相应内置方法，这样我们就能够使用诸如 `+` `-` 等内置的通用语法

举个例子：当你对两个用户定义的类的实体使用加法时，它们实际上会去调用 `__add__()` 或 `__radd__()`

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)

>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)

>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)
```

Type Dispatching

```
def __add__(self, other):
    if isinstance(other, int):
        n = self.numer + self.denom * other
        d = self.denom
    elif isinstance(other, Ratio):
        n = self.numer * other.denom + self.denom * other.numer
        d = self.denom * other.denom
    elif isinstance(other, float):
        return float(self) + other
    g = gcd(n, d)
    return Ratio(n//g, d//g)
```

每次使用加法的时候都要判断一堆的`if isinstance`，这是Python为什么慢的原因之一

Type Convertion

```
elif isinstance(other, float):
    return float(self) + other
```

Lecture 21: Composition

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c9b69703-d820-4534-853a-606e2ec3f563/21-Composition_1pp.pdf

composition是一节习题课

Linked Lists

链表如何用Python的object来表示呢?

A linked list is either empty or a first value and the rest of the linked list

Linked list class: attributes are passed to __init__

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Some zero-length sequence

Returns whether rest is a Link

Linked List Processing

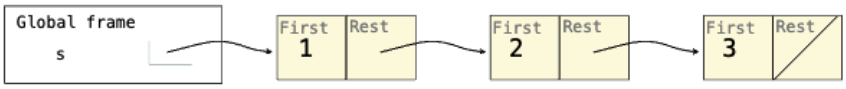
```
square, odd = lambda x: x * x, lambda x: x % 2 == 1
list(map(square, filter(odd, range(1, 6))))           # [1, 9, 25]
map_link(square, filter_link(odd, range_link(1, 6)))  # Link(1, Link(9, Link(25)))

def range_link(start, end):
    """Return a Link containing consecutive integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """

def map_link(f, s):
    """Return a Link that contains f(x) for each x in Link s.

    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """

def filter_link(f, s):
    """Return a Link that contains only the elements x of Link s for which f(x)
    is a true value.

    >>> filter_link(odd, range_link(3, 6))
    Link(3, Link(5))
    """
```

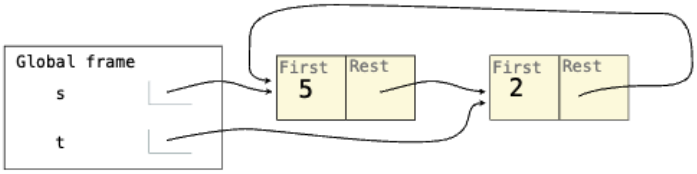
Linked List Mutation

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```





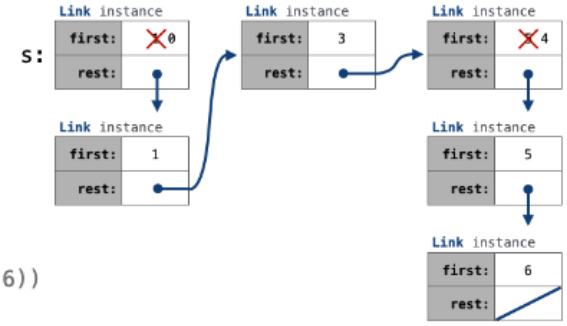
Note: The actual environment diagram is much more complicated.

给一个object命名只是起了个别名，并不会产生一份新的copy

Linked List Mutation Example

Adding to a Set Represented as an Ordered List

```
def add(s, v):
    """Add v to s, returning modified s."""

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))
    """
    assert s is not List.empty
    if s.first > v:
        s.first, s.rest = ____v____ , ____Link(s.first, s.rest)____
    elif s.first < v and empty(s.rest):
        s.rest = ____Link(v)____
    elif s.first < v:
        ____add(s.rest, v)____
    return s
```



16

Tree Class

面向对象编程是实现ADT（数据类型抽象）的一种方式

Pass

Tree Mutation

Pass

Week 9

Lecture 22: Efficiency

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/32df0169-80f3-4e98-9b72-e7bfb3e7a14f/22-Efficiency_1pp.pdf

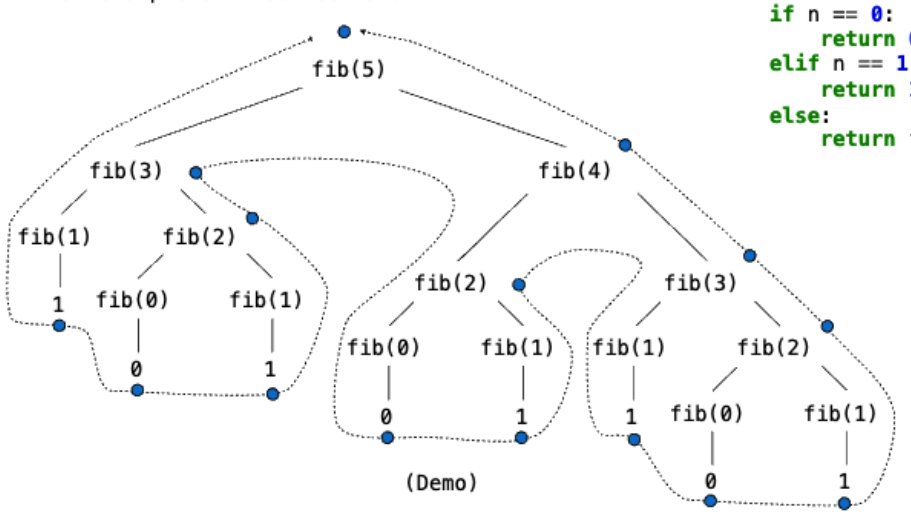
---

Measuring Efficiency

考虑斐波那契函数的复杂度，可以看到重复了很多次

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:




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

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

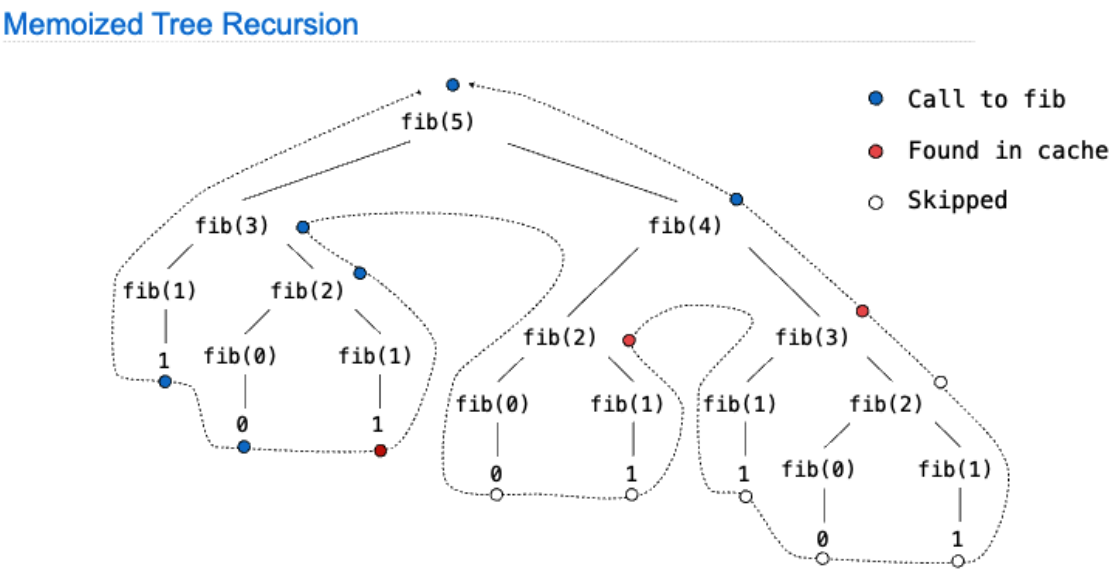
4

Memoization

就是把重复的计算结果记录下来，再次遇到时就可以直接得到结果

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
```

这样会加速很多



Exponentiation

快速幂

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

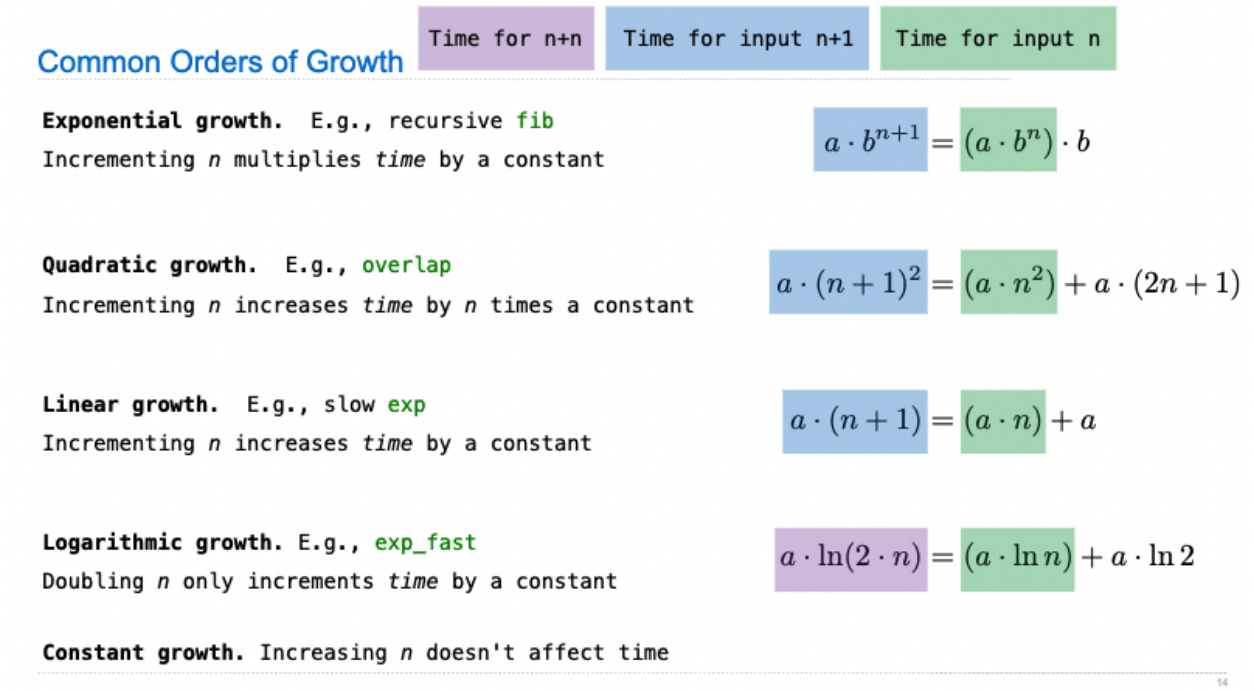
$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

def square(x):
    return x * x
```

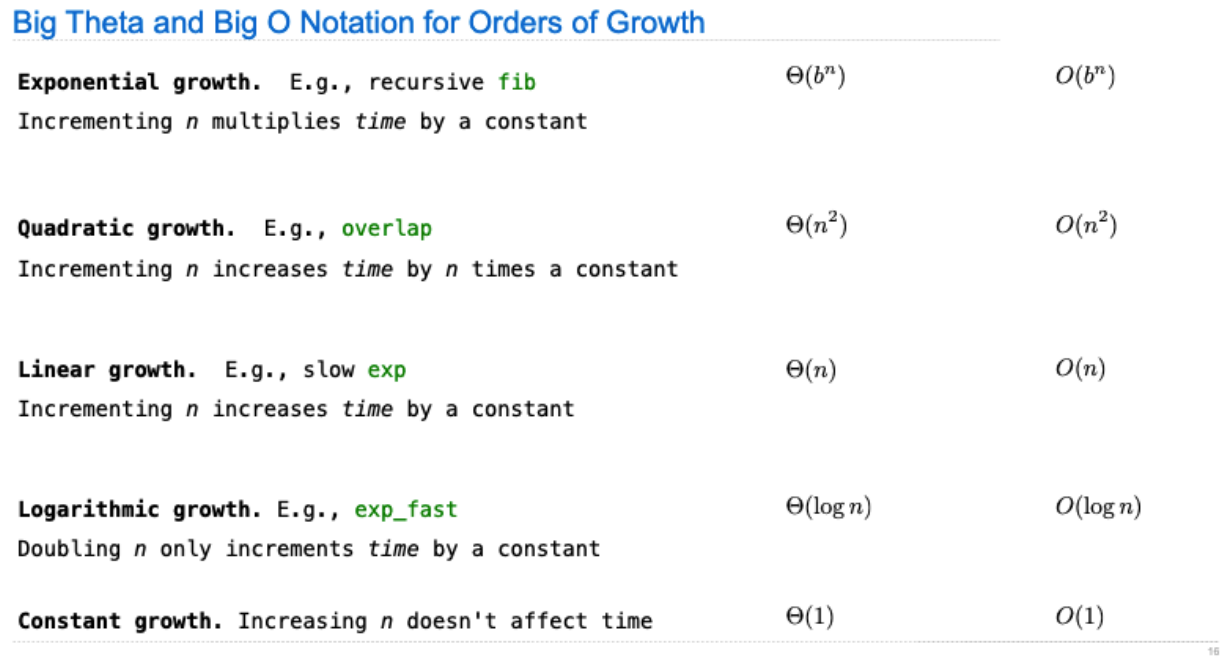
$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

(Demo)

Orders of Growth



Order of Growth Notation



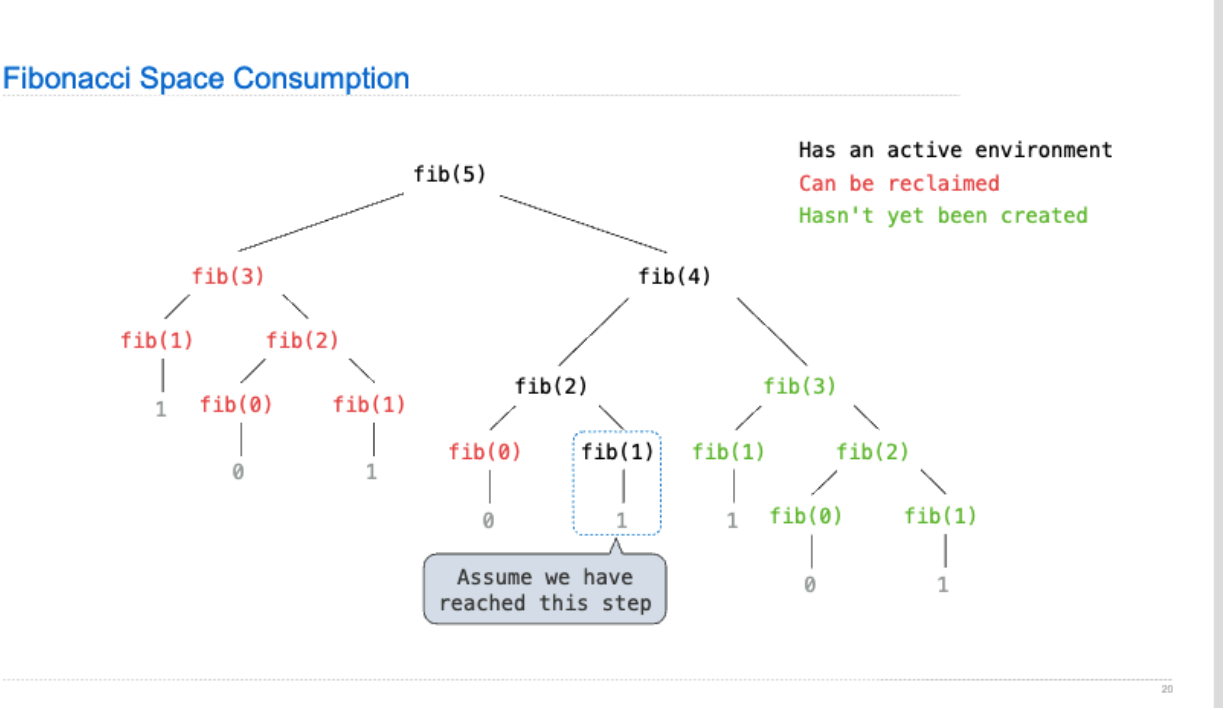
Space

在Python中，只有**活跃的环境**才会持续存在，消耗内存

那么什么是活跃的环境呢？

- 正在被调用的函数所在的环境，函数`return`之后这个`frame`就会被删除掉，就不是`active`了
- 活跃的环境的父代环境（`parent environments`）

例子：



Lecture 23: Decomposition

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c3b7043e-3845-4f94-a501-782a15f48599/23-Decomposition_1pp.pdf

Modular Design

写程序的时候要注意模块化设计

- 其中一个设计原则就是要把程序中不同功能的部分分开，也就是说你修改某个部分的代码之后其他的组件应该不受影响
- 每个隔离开来的模块都必须能够被单独的测试

Example： Restaurant Search

Pass

Example: Similar Restaurants

Pass

Example： Reading Files

Pass

Set Intersection

双指针算法计算两个序列的重合度

Pass

Sets

set 是Python内置的一种容器类型

- 使用大括号表示
- 和数学里的集合一样，不可以有重复的元素
- sets是无序的

```
>>> s = {'one', 'two', 'three', 'four', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
>>> 'three' in s
True
>>> len(s)
4
>>> s.union({'one', 'five'})
{'three', 'five', 'one', 'four', 'two'}
>>> s.intersection({'six', 'five', 'four', 'three'})
{'three', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
```

Lecture 24： Data Examples

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/de1f9280-e930-49c2-9531-a5deaebae5ed/24-Data_Examples_1pp.pdf

---

Examples: Objects

```
Pass
```

Examples: Iterables & Iterators

```
Pass
```

Examples: Linked Lists

```
Pass
```

Week11

Lecture 27: Scheme

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/55023718-cafe-4cc7-8e2b-9b0e96e39cf6/27-Scheme_1pp.pdf

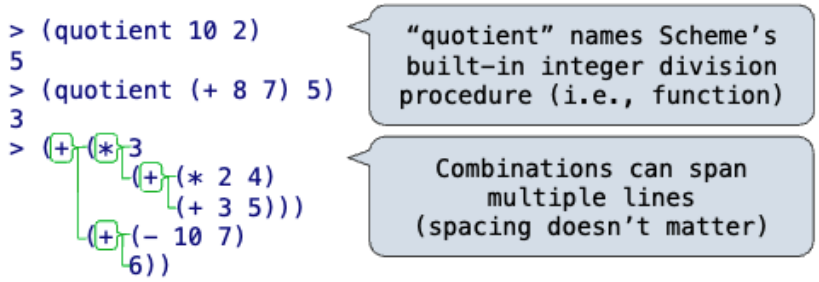
Scheme

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

Numbers are self-evaluating; symbols are bound to values

Call expressions include an operator and 0 or more operands in parentheses



缩进与空格只是方便人类阅读

Special Forms

A combination that is not a call expression is a special form:

- **if** expression: (if <predicate> <consequent> <alternative>)
- **and** and **or**: (and <e1> ... <en>), (or <e1> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative

```
> (define pi 3.14)
> (* pi 2)
6.28

> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

The symbol "pi" is bound to 3.14 in the global frame

A procedure is created and bound to the symbol "abs"

(Demo)

Scheme Interpreters

我们学习scheme的目的是为了用Python建立一个scheme的解释器

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

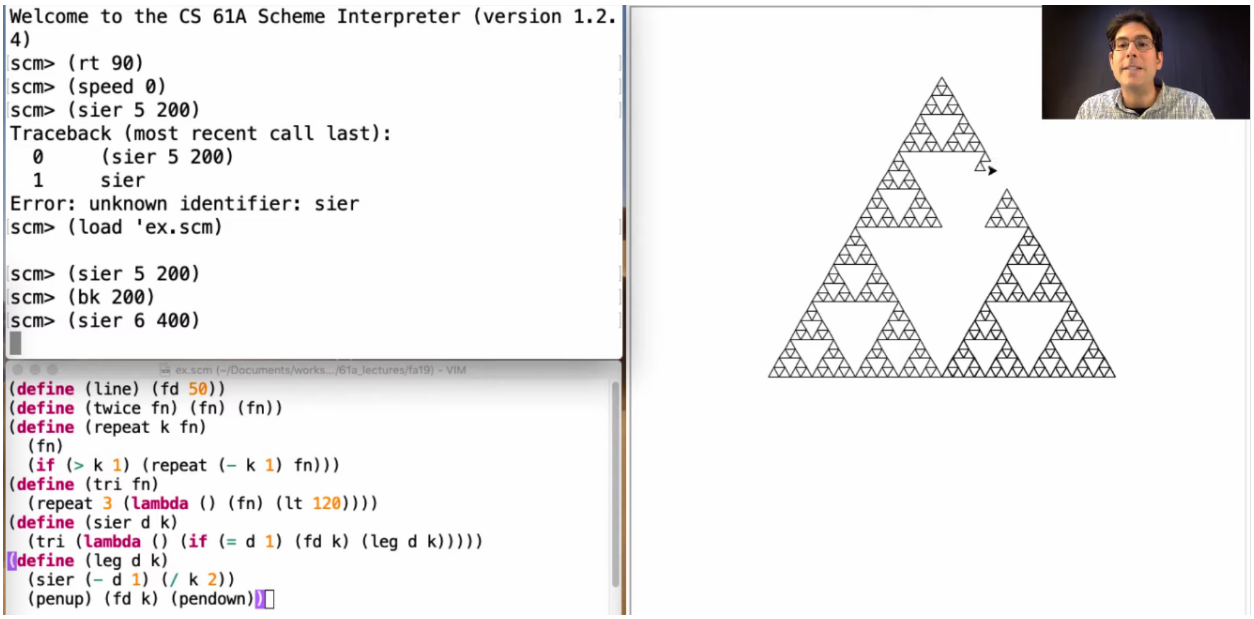
An operator can be a call expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

▶ 12

Evaluates to the $x+y+z^2$ procedure

Example: Sierpinski's Triangle



More Special Forms

Cond 相当于 Python中的 `if elif else` 结构

begin 可以把好几个语句结合起来，按顺序一起执行

Cond & Begin

The cond special form that behaves like if-elif-else statements in Python

```
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```
(cond ((> x 10) (print 'big))
      ((> x 5)  (print 'medium))
      (else     (print 'small)))
```

```
(print
  (cond ((> x 10) 'big)
        ((> x 5)  'medium)
        (else     'small)))
```

The begin special form combines multiple expressions into one expression

```
if x > 10:
    print('big')
    print('guy')
else:
    print('small')
    print('fry')
```

```
(cond ((> x 10) (begin (print 'big)   (print 'guy)))
      (else     (begin (print 'small) (print 'fry))))
```

```
(if (> x 10) (begin
                (print 'big)
                (print 'guy))
             (begin
                (print 'small)
                (print 'fry)))
```

Let 就是只作用于一句话的define，用完即弃不用考虑作用于问题

使用例子如下

Let Expressions

The let special form binds symbols to values temporarily; just for one expression

```
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
a and b are still bound down here
```

```
(define c (let ((a 3)
                (b (+ 2 2)))
               (sqrt (+ (* a a) (* b b)))))
a and b are not bound down here
```

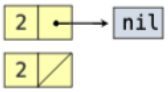
Lists

就像Python有自己的内置的数据结构一样，scheme也有，也就是这里要介绍的Linked List

In the late 1950s, computer scientists used confusing names

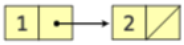
- **cons:** Two-argument procedure that creates a linked list
- **car:** Procedure that returns the first element of a list
- **cdr:** Procedure that returns the rest of a list
- **nil:** The empty list

(cons 2 nil)



Important! Scheme lists are written in parentheses with elements separated by spaces

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```



Symbolic Programming

`'` 有两个作用：

- `'` 把其之后的内容当做符号显示，而不是立刻 evaluate 这个内容
- 在 `()` 前面表示组成list

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

(Demo)

Programs as Data

由于上文中提到的symbolic programming 我们就可以使用scheme来生成scheme的代码了

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)

scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

例子如下：

这样就可以使用scheme生成“用于计算阶乘的scheme程序”

```
~/lec$ ./scheme -i ex.scm
Welcome to the CS 61A Scheme Interpreter (version 1.2.2)

scm> fact
(lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
scm> (fact 3)
6
scm> (fact 5)
120
scm> ^D
~/lec$ ./scheme -i ex.scm
Welcome to the CS 61A Scheme Interpreter (version 1.2.2)

scm> (fact 5)
120
scm> (fact-exp 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
scm> (eval (fact-exp 5))
120
```

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))

(define (fact-exp n)
  (if (= n 0) 1 (list '* n (fact-exp (- n 1)))))
```

Generating Code

Quasiquotation

There are two ways to quote an expression

```
  Quote:      '(a b)   =>   (a b)

  Quasiquote: `(a b)   =>   (a b)
```

They are different because parts of a quasiquoted expression can be unquoted with ,

```
              (define b 4)

  Quote:      '(a ,(+ b 1))  =>   (a (unquote (+ b 1))

  Quasiquote: `(a ,(+ b 1))  =>   (a 5)
```

Quasiquotation is particularly convenient for generating Scheme expressions:

```
              (define (make-add-procedure n) `(lambda (d) (+ d ,n)))

              (make-add-procedure 2)  => (lambda (d) (+ d 2))
```

`,` 的意思是unquote它之后的内容

quasiquote `` ` `` 和 `,` 一起用就会 eval `,` 之后的内容

Example: While statements

scheme没有while语句，所以只能使用递归来实现迭代的效果，如下图：

What's the sum of the squares of even numbers less than 10, starting with 2?

```
x = 2
total = 0
while x < 10:
    total = total + x * x
    x = x + 2
```

```
(begin
  (define (f x total)
    (if (< x 10)
      (f (+ x 2) (+ total (* x x)))
      total))
  (f 2 0)))
```

What's the sum of the numbers whose squares are less than 50, starting with 1?

```
x = 1
total = 0
while x * x < 50:
    total = total + x
    x = x + 1
```

```
(begin
  (define (f x total)
    (if (< (* x x) 50)
      (f (+ x 1) (+ total x))
      total))
  (f 1 0)))
```

Lecture 28: Exceptions

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/dd2a4be4-2482-4032-a112-56a48ed9bc63/28-Exceptions_1pp.pdf

Exception

有的时候程序会出现异常行为

- 数据传输中网络突然丢失连接
- 传递了错误的函数参数的数据类型

在Python中有Exception（异常）机制。这是编程语言内置（bulit-in）的一种用来表达以及应对特殊情况（exceptional conditions）的机制。

当错误发生的时候，Python会 `raises` 一个 `exception`

Exceptions 是可以被程序所处理的，这样就可以防止Python的解释器因为错误而停机（halting）

但是如果exception没有被你写的程序处理，那么Python解释器会停机，并打印出 a stack trace（错误信息）

掌握exception需要注意的点

- exceptions 是对象！ 它有相应的构造器（constructors）
- 它允许了 non-local continuations of control:

  f, g, h都是函数。如果f调用g，g调用h，exceptions 可以使得程序执行的流程从h直接改到f，跳过g的return
- 注意！exception是很慢的，不要滥用。

Raising Exceptions

Assert 语句抛出一个 `AssertionError` 类型的异常

语法：`assert <expression>, <string>` ;当 <expression> 是 `False` 的时候，就会抛出异常，<string> 的内容作为提示信息

Assertions可以相对的随意的使用。因为它是可以被忽略从而提升Python效率的，我们只需要 `python3 -O` 就可，O是‘optimized’的意思

assertions是否被启用可以通过 `__debug__` 来监控

异常可以通过 `raise` 语句来抛出

语法：`raise <expression>` ； `<expression>` 必须是基础异常的子类或者是其中的一个实例才可以。

构建异常就像构建其他对象一样。例如： `TypeError('Bad argument!')`

常见的异常：

- `TypeError`: A function is passed the wrong number/type of argument
- `NameError`: A name wasn’t found
- `KeyError`: A key wasn’t found in a dictionary
- `RuntimeError`: Catch-all for troubles during interpretation

```
>>> raise TypeError('Bad argument')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Bad argument
>>> abs('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
>>> hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
>>> {}['hello']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'hello'
```

Try Statements

try statement 可以处理异常

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Execution rule:

The <try suite> is executed first.

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception.

例子：

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

多重的try语句：expect语句会和距离它最近的，引起expect异常的那个try语句配对

Example: Reduce

```
pass
```

Lecture 29: Calculator

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ec2591dc-9ce8-44fd-9333-1cd02b6d39f4/29-Calculator_1pp.pdf

Programing Languages

A computer typically executes programs written in many different programming languages

Machine languages: statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

High-level languages: statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

Python 3

```
def square(x):
    return x * x
```

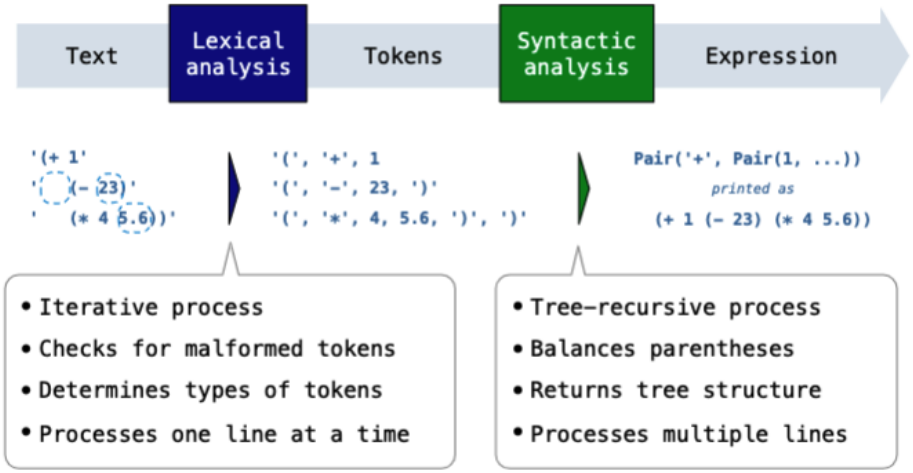
```
from dis import dis
dis(square)
```

Python 3 Byte Code

```
LOAD_FAST                0 (x)
LOAD_FAST                0 (x)
BINARY_MULTIPLY
RETURN_VALUE
```

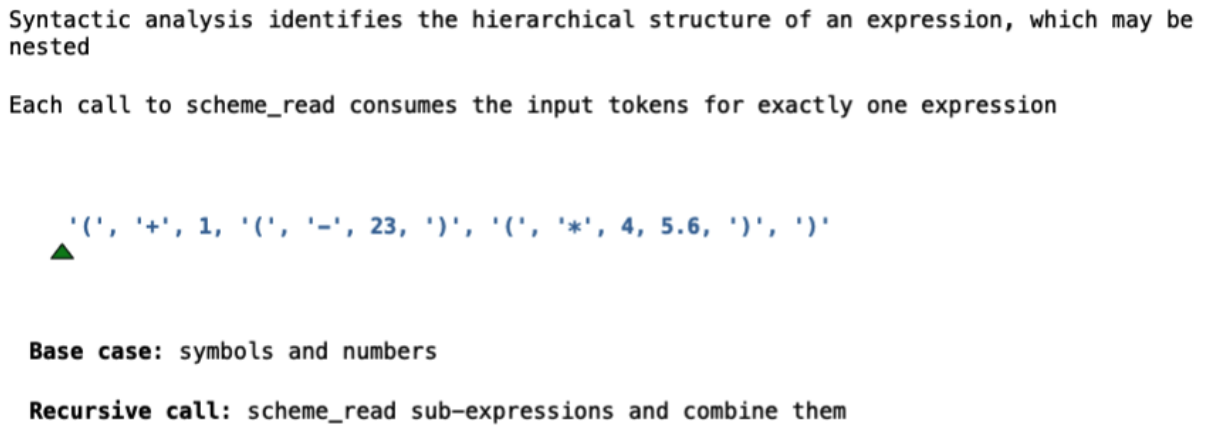
Parsing

A Parser takes text and returns an expression



parsing主要是做两个工作：

- 词法分析：检查代码，把它们化为单个的token
- 句法分析：在词法分析得出的tokens的基础上把代码变为更规律的表达式



词法分析会分析表达式的结构层次，这个时候就会遇到nested的表达式，由于Scheme的左右括号是平衡的，所以我们的解决办法就是使用递归来处理

Calculator

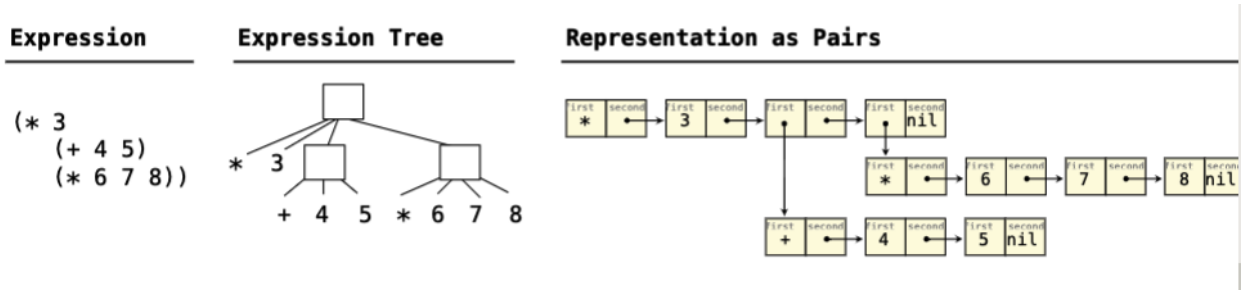
这里的计算器是指Scheme-Syntax-Calculator，也就是由Python写出来的scheme语法分析器

由于scheme可以用list这一种数据结构来表示内部的所有逻辑，运算，函数调用等，所以我们使用 `Pair` 这个类来表示scheme

```
class Pair:
    """A Pair has two instance attributes:
    first and second.

    For a Pair to be a well-formed list,
    second is either a well-formed list or nil
    Some methods only apply to well-formed lis
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Calculator Syntax

这个语法分析器有源语和调用，可以表示成下面这三种形式：



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

Call: A call expression evaluates to its argument values combined by an operator.

+: Sum of the arguments

*: Product of the arguments

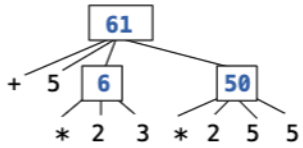
-: If one argument, negate it. If more than one, subtract the rest from the first.

/: If one argument, invert it. If more than one, divide the rest from the first.

Expression

```
(+ 5
   (* 2 3)
   (* 2 5 5))
```

Expression Tree



Evaluation

The Eval Function

The eval function computes the value of an expression, which is always a number

It is a generic function that dispatches on the type of the expression (primitive or call)

Implementation

```
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

Recursive call returns a number for each operand

'+', '-', '*', '/'

A Scheme list of numbers

Language Semantics

A number evaluates...

to itself

A call expression evaluates...

to its argument values combined by an operator

Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values

In calculator, all operations are named by built-in operators: +, -, *, /

Implementation

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

Language Semantics

```
+:
   Sum of the arguments
-:
   ...
...
```

(Demo)

Interactive Interpreters

Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
4. **Evaluate** the expression
5. If any errors occur, report those errors, otherwise
6. **Print** the value of the expression and repeat

Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
- **Apply:** No arguments to - raises TypeError("- requires at least 1 argument")

Handling Exceptions

An interactive interpreter prints information about each error

A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment

---

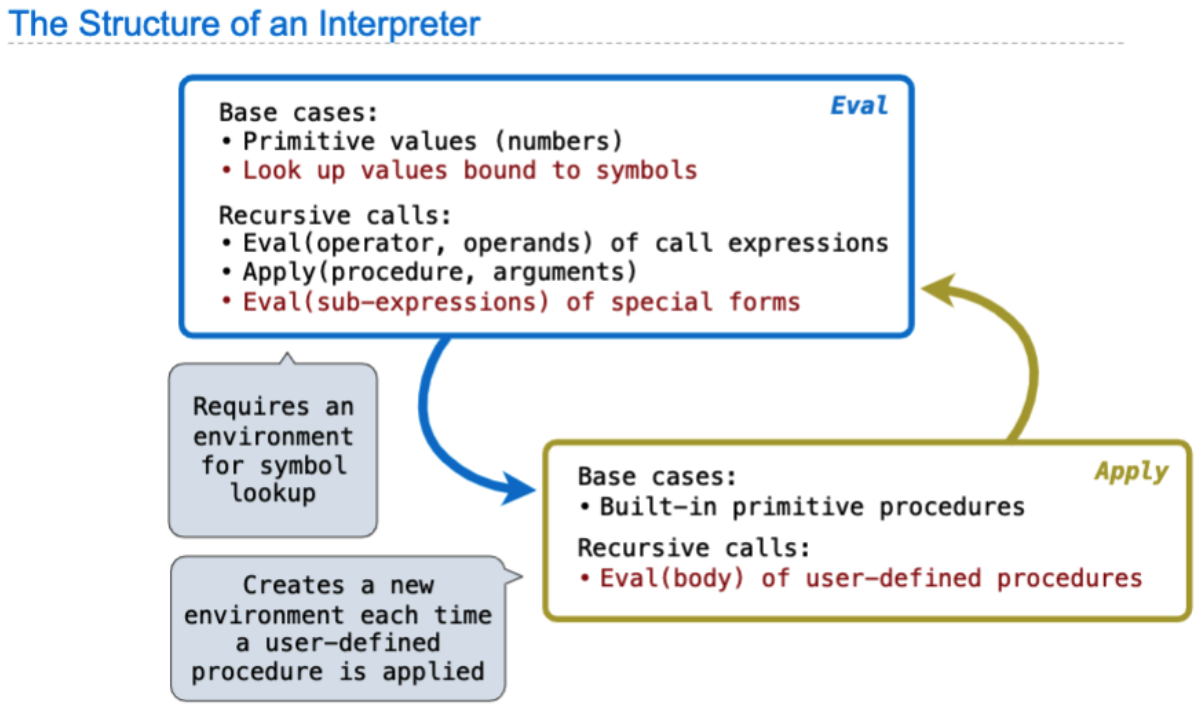
Week12

Lecture 30: Interpreters

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/72c4fa7e-6ad2-4f1b-8053-00187c467bd3/30-Interpreters_1pp.pdf

Interpretering Scheme

介绍scheme解释器的结构



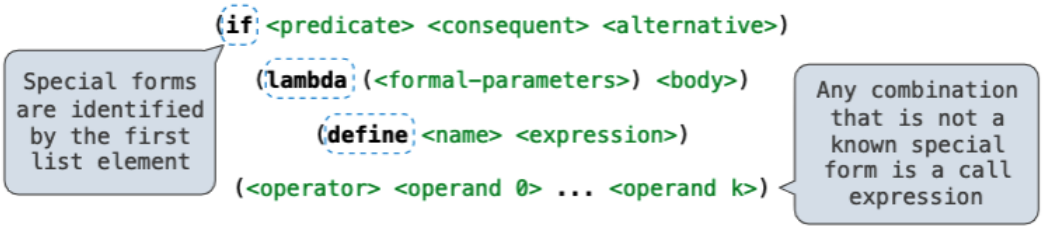
Special Forms

上文中提到的Scheme eval函数会区分不同的表达式：

Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations




```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))

(demo (list 1 2))
```

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e1> ... <en>), (**or** <e1> ... <en>)
- **Cond** expression: (**cond** (<p1> <e1>) ... (<pn> <en>) (else <e>))

The value of an if expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

do_if_form

在我们的这个解释器里，eval逻辑表达式只会eval其部分内容（sub-expression）

Quotation

Quotation

The quote special form evaluates to the quoted expression, which is not evaluated

(quote <expression>) (quote (+ 1 2)) evaluates to the three-element Scheme list (+ 1 2)

The <expression> itself is the value of the whole quote expression

'<expression> is shorthand for (quote <expression>)

(quote (1 2)) is equivalent to '(1 2)

The scheme_read parser converts shorthand ' to a combination that starts with quote

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

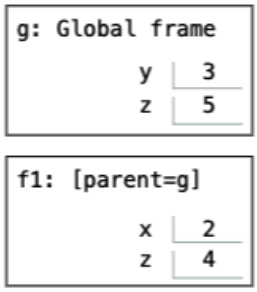
```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals ····· A scheme list of symbols
        self.body = body ····· A scheme list of expressions
        self.env = env ····· A Frame instance
```

Frames and Environments

A frame represents an environment by having a parent frame

Frames are Python instances with methods **lookup** and **define**

In Project 4, Frames do not hold return values



(Demo)

Define Expressions

自己定义的函数和定义变量是一样的，只是定义函数相当于给一个lambda函数绑定别名

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>
2. Bind <name> to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression

```
(define (<name> <formal parameters>) <body>)
```

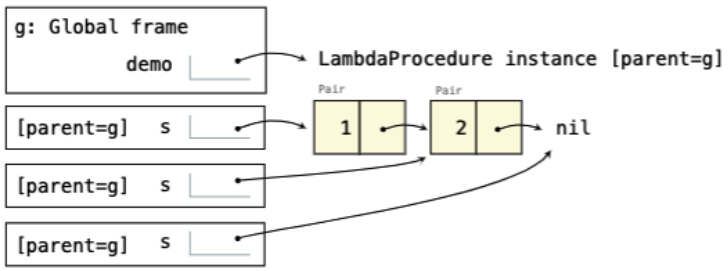
```
(define <name> (lambda (<formal parameters>) <body>))
```

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** attribute of the procedure

Evaluate the body of the procedure in the environment that starts with this new frame

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
(demo (list 1 2))
```

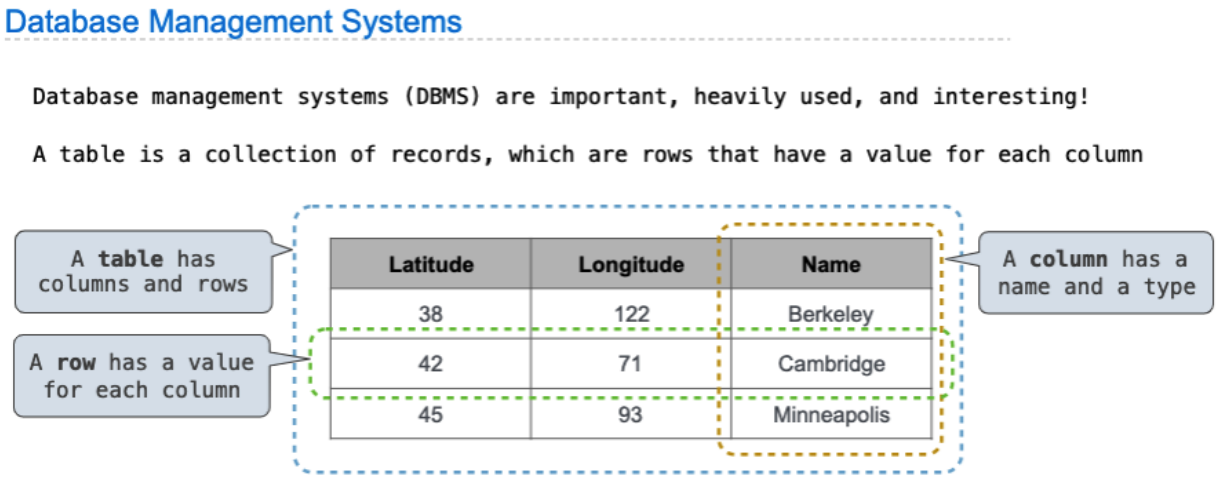


Lecture 31： Declarative Programming

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/95e74a37-f54f-46c7-9c1e-8df60fc5a38c/31-Declarative_Programming_1pp.pdf

Declarative Languages

前面我们已经介绍过函数式编程，面向过程编程，面向对象编程了，接下来我们介绍声明式编程。SQL就是一种声明式编程语言。



声明式编程和命令式编程的区别：

In **declarative languages** such as SQL & Prolog:

- A "program" is a description of the desired result
- The interpreter figures out how to generate the result

In **imperative languages** such as Python & Scheme:

- A "program" is a description of computational processes
- The interpreter carries out execution/evaluation rules

Structured Query Language(SQL)

SQL Overview

The SQL language is an ANSI and ISO standard, but DBMS's implement custom variants

- A **select** statement creates a new table, either from scratch or by projecting a table
- A **create table** statement gives a global name to a table
- Lots of other statements exist: **analyze**, **delete**, **explain**, **insert**, **replace**, **update**, etc.
- Most of the important action is in the **select** statement

Selecting Value Literals

A **select** statement always includes a comma-separated list of column descriptions

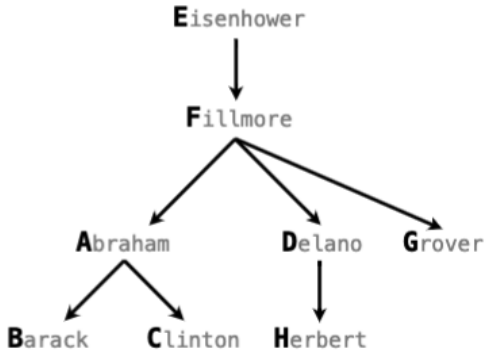
A column description is an expression, optionally followed by **as** and a column name

```
select [expression] as [name], [expression] as [name]; ...
```

Selecting literals creates a one-row table

The union of two select statements is a table
containing the rows of both of their results

```
select "delano" as parent, "herbert" as child;union
select "abraham"         , "barack"           union
select "abraham"         , "clinton"          union
select "fillmore"        , "abraham"          union
select "fillmore"        , "delano"           union
select "fillmore"        , "grover"           union
select "eisenhower"      , "fillmore";
```



Naming Tables

SQL is often used as an interactive language

The result of a **select** statement is displayed to the user, but not stored

A **create table** statement gives the result a name

```
create table [name] as [select statement];
```

```
create table parents as
select "delano" as parent, "herbert" as child union
select "abraham"         , "barack"           union
select "abraham"         , "clinton"          union
select "fillmore"        , "abraham"          union
select "fillmore"        , "delano"           union
select "fillmore"        , "grover"           union
select "eisenhower"      , "fillmore";
```

Parents:

Parent	Child
abraham	barack
abraham	clinton
delano	herbert
fillmore	abraham
fillmore	delano
fillmore	grover
eisenhower	fillmore

只用select语句并不会创建新的表，而是创建了视图

Projecting Tables

Select Statements Project Existing Tables

A **select** statement can specify an input table using a **from** clause

A subset of the rows of the input table can be selected using a **where** clause

An ordering over the remaining rows can be declared using an **order by** clause

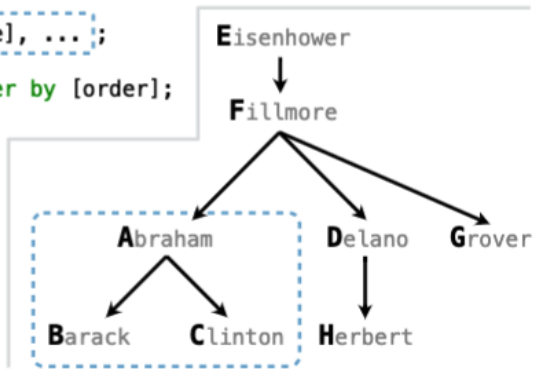
Column descriptions determine how each input row is projected to a result row

```
select [expression] as [name], [expression] as [name], ...;
select [columns] from [table] where [condition] order by [order];
select child from parents where parent = "abraham";
select parent from parents where parent > child;
```

Child
barack
clinton

Parent
fillmore
fillmore

(Demo)



Arithmetic

注意你建表时候的行的顺序不一定就是真实存储的顺序，真实存储的顺序取决度数据库的引擎

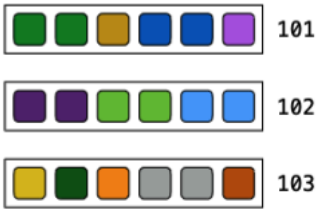
Arithmetic in Select Expressions

In a select expression, column names evaluate to row values

Arithmetic expressions can combine row values and constants

```
create table lift as
  select 101 as chair, 2 as single, 2 as couple union
  select 102         , 0          , 3           union
  select 103         , 4          , 1;

select chair, single + 2 * couple as total from lift;
```

chair	total
101	6
102	6
103	6



Discussion Question

Given the table **ints** that describes how to sum powers of 2 to form various integers

```
create table ints as
  select "zero" as word, 0 as one, 0 as two, 0 as four, 0 as eight union
  select "one"         , 1       , 0       , 0        , 0          union
  select "two"         , 0       , 2       , 0        , 0          union
  select "three"       , 1       , 2       , 0        , 0          union
  select "four"        , 0       , 0       , 4        , 0          union
  select "five"        , 1       , 0       , 4        , 0          union
  select "six"         , 0       , 2       , 4        , 0          union
  select "seven"       , 1       , 2       , 4        , 0          union
  select "eight"       , 0       , 0       , 0        , 8          union
  select "nine"        , 1       , 0       , 0        , 8;
```

(A) Write a select statement for a two-column table of the **word** and **value** for each integer

word	value
zero	0
one	1
two	2
three	3
...	...

(B) Write a select statement for the **word** names of the powers of two

word
one
two
four
eight

(Demo)

Week 13

Lecture 32: Tables

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ad3a79a3-b925-4568-be35-5669ef4c3503/32-Tables_1pp.pdf

Joining Tables

语法：

```
select needed_col from table1, table2
        where col1 = col2 and col3 = condition

例子如下：

select parent from parents, dogs
        where child = name and fur = "curly"
```

needed_col: 需要显示的列
table1, table2: 需要合并的表
col1, col2, col3: 条件所限制的列
condition: 列所需的条件

select语句的一般形式：

```
        [expression] as [name], [expression] as [name], ...
select [columns] from [table] where [expression] order by [expression];
```

Aliases and Dot Expressions

使用 `.` 和别名可以允许我们合并同一张表

```
Select all pairs of siblings

  SELECT a.child AS first, b.child AS second
    FROM parents AS a, parents AS b
    WHERE a.parent = b.parent AND a.child < b.child;
```

例子：

```
SELECT grandog FROM grandparents, dogs AS c, dogs AS d
                WHERE grandog = c.name AND
                      granpup = d.name AND
                      c.fur = d.fur;
```

Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
[expression] AS [name], [expression] AS [name], ...
SELECT [columns] FROM [table] WHERE [expression] ORDER BY [expression];
```

Combine values: +, -, *, /, %, and, or

Transform values: abs, round, not, -

Compare values: <, <=, >, >=, <>, !=, =

String Expressions

String values can be combined to form longer strings

```
sqlite> SELECT "hello," || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python

```
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
sqlite> SELECT substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1) FROM phrase;
low
```

Strings can be used to represent structured values, but doing so is rarely a good idea

```
sqlite> CREATE TABLE lists AS SELECT "one" AS car, "two,three,four" AS cdr;
sqlite> SELECT substr(cdr, 1, instr(cdr, ",")-1) AS cadr FROM lists;
two
```

Lecture 33: Aggregation

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4d20bc5e-6bc4-4ec6-a4e3-857b0056392e/33-Aggregation_1pp.pdf

---

Aggregation

select的一般形式：

```
        [expression] as [name], [expression] as [name], ...
select [columns] from [table] where [expression] order by [expression];
```

当聚合函数作用在 [columns] 语句时，它会计算出这一列某些行数据（也就是作用在group上的）的某个值（如min, max, avg等）

常用的聚合函数：

- max()
- min()
- avg()
- count()
- distinct()

```
An aggregate function also selects some row in the table to supply the values of columns
that are not aggregated.  In the case of max or min, this row is that of the max or min
value.  Otherwise, it is arbitrary.
```

Groups

语法：

```
                [expression] as [name], [expression] as [name], ...
select [columns] from [table] group by [expression] having [expression];
```

```
The number of groups is the number of unique values of an expression

A having clause filters the set of groups that are aggregated
```

例子：

```
select weight/legs, count(*) from animals group by weight/legs having count(*)>1;
```

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

weight/legs=5
weight/legs=2
weight/legs=2
weight/legs=3
weight/legs=5
weight/legs=6000

weight/legs	count(*)
5	2
2	2

Lecture 34: Databases

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e6c9cbb6-6d15-4168-afd8-3440c8d045e8/34-Databases_1pp.pdf

Week 14

Lecture 35: Tail Calls

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6229d7a1-ce0a-4803-a621-d4e06c05ba73/35-Tail_Calls_1pp.pdf

In Python, recursive calls always create new active frames

`factorial(n, k)` computes: $n! * k$

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

	Time	Space
	$\Theta(n)$	$\Theta(n)$
	$\Theta(n)$	$\Theta(1)$

由于Python中调用函数会新建frame，所以递归函数的空间复杂度是 $O(n)$。但是循环不需要新建frame，所以空间复杂度是 $O(1)$。Tail Recursion可以解决这个问题。

举个例子：如下图，我们递归到递归边界的时候，会返回240这个结果。这个时候所有之前递归调用函数产生的frame也只是会接收到这个返回值240，然后再返回给parent frame。注意这个时候每个老frame里的 `n, k` 都是不会再被用到的，它们所占用的空间也是被浪费掉的。Tail Recursion就是根据这个来进行优化的。

Tail Calls

Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls using only a constant amount of space.

A tail call is a call expression in a tail context:

- The last body sub-expression in a **lambda** expression
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and, or, begin,** or **let**

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
                 (* k n))))
```

接下来举个tail call 优化的例子，如下图，`length` 不是tail call，因为再次调用length之后还有 `+1` ，不是直接返回返回值即可，原来的frame还需要保留。解决方案就是定义一个递归的helper函数，使得 `+1` 的操作在传参数的时候完成。

Example: Length of a List

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

Not a tail context

A call expression is not a tail call if more computation is still required in the calling procedure

Linear recursive procedures can often be re-written to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

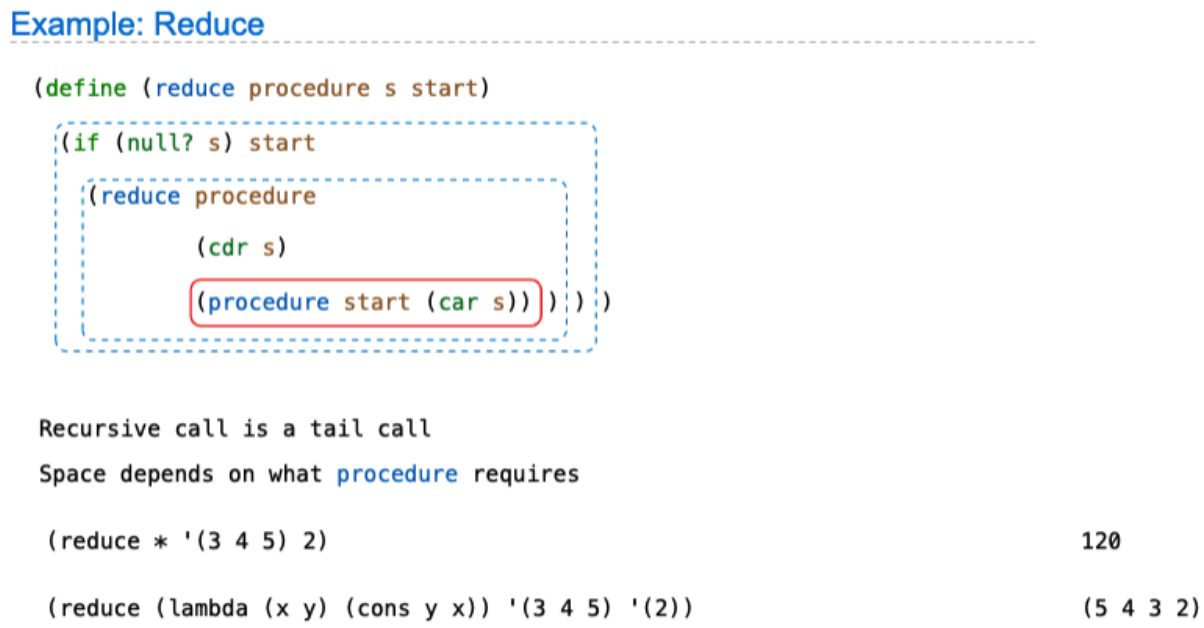
Recursive call is a tail call

Tail Recursion Examples

pass

Map and Reduce

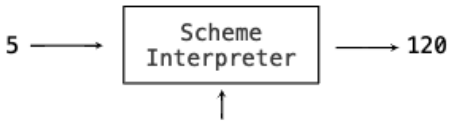
下面这个例子，reduce是一个高阶函数，它是不是tail call取决于prcedure这个函数是不是只用 $O(1)$的空间



General Computing Machines

Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine




```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of evaluation rules

Week 15

Lecture 35: Macros

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/910d9e79-14be-4342-ade3-754c9ba028da/36-Macros_1pp.pdf

宏对于解释器来说是这样被处理的

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions *without evaluating them first*
- Evaluate the expression returned from the macro procedure

Leture 36: Final Examples

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3104c850-2d21-4b30-95c6-51df53201e4b/37-Final_Examples_1pp.pdf

How to Design Programs

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function's purpose.

Function Template
Translate the data definitions into an outline of the function.

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.