UCB CS61A Lecture Notes

Xiao Fan 收录于 lecture notes

2021-03-17 2021-03-17 约 9403 字 预计阅读 19 分钟

1 Introduction

本课程基于*Structure and Interpretation of Computer Programs* (SICP)。课程网址: https://inst.eecs.berkeley.edu/~cs61a/fa20/

1.1 Python features

doctests

在python的 `def` 关键词下的一行用 `"""` 包裹的文字是叫做 `docstring` ，用来解释这个函数所做的事情。在 `docstring` 中如果加入了 `>>>` ，这就是 `doctest` ，用来测试函数的正确性，比如

Python

```
# ex.py
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D
    >>> q, r = divide_exact(2013,10)
    >>> q
    201
    >>> r
    3
    """
    return floordiv(n, d), mod(n, d)
```

在shell中运行

Bash

```
python3 -m doctest ex.py
```

如果没有任何输出，说明结果正确，即q确实是201，r是3

assert

Python中的 `assert` 语法为后面跟一个condition，如果这个condition为false，则打印 `AssertionError:<some thing>` ，比如：

C

```
assert a > 0, 'a must greater than 0'
```

lambda function

返回一个值为 `function` 的表达式，表达式中没有 `return` 关键字

Python

```
square = lambda x: x * x
# square is a function that takes in x and returns x * x
```

逆函数的实现：

Python

```
def search(f):
    x = 0
    while not f(x):
        x + =1
    return x
def inverse(f):
    """return g(y) such that g(f(x)) = x"""
    return lambda y: search(lambda x: f(x) == y)
```

conditional expression

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

先判断 `<predicate>` ，如果为true，整个表达式的值就是 `<consequent>` 的值，否则是 `<alternative>` 的值。

decorator

`@` 是一个decorator，其作用是后面跟一个函数 `fn1` ，放在另一个函数 `fn2` 上方，等效于调用 `fn2 = fn1(fn2)`

Python

```
def trace1(fn):
    def traced(x):
        print('Calling', fn, 'on argument', x)
        return fn(x)
    return traced

@trace1
def square(x):
    return x * x

>>> square(2)
Calling <function square at 0x1006ee170> on argument 2
4
```

1.2 Higher order function

higher order function

输入或者输出其中一个为函数的函数就是高次函数，比如

Python

```
def apply_twice(f, x):
    return f(f(x))
def square(x):
    return x * x
```

```
>>> apply_twice(square, 3)
>>> 81
```

Python

```
def make_adder(n):
    def ader(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)  # equals 7
```

注意， `add_three` 这个函数是在 `make_adder` 中被创建的，因此它的parent frame是 `make_adder` ，因此可以访问到这个parent frame环境中的所有本地变量，包括n这个等于3的变量

self reference

函数可以返回自身，从而做到连续调用

Python

```
def print_sums(x):
    print(x)
    def next_sum(y):
        return print_sums(x+y)
    return next_sum

>>> print_sums(1)(3)(5)
1
4
9
```

curry

柯里化：将一个需要接受多个参数的函数转换为接受一个参数并且返回**[一个可以接受剩余参数并返回值的函数]**的函数

3个 `def` 的嵌套可以用于构造一个构造函数

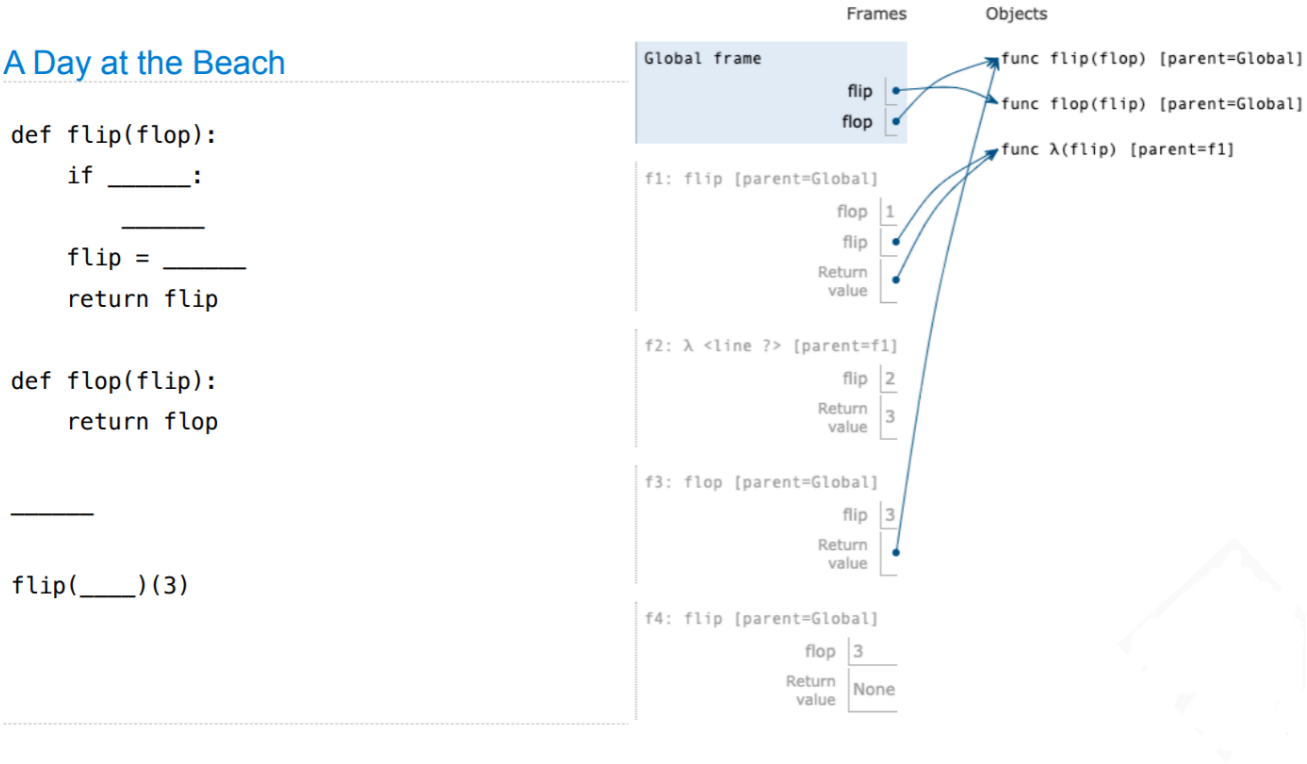
Python

```
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g

>>> m = curry2(add)
>>> add_three = m(3)
>>> add_three(2)
5
```

```
# 效果等同于
>>> curry = lambda f: lambda x: lambda y: f(x, y)
```

1.3 Environment Diagram



在global frame中，flip和flop分别被赋值给了 `func flip` 和 `func flop` ，因此第4个空格应该是

Python

```
flip, flop = flop, flip
```

f1 frame是flip，但是注意此时 `func flip` 在global frame中实际上是flop，因此第5个空格中应该是先调用了flop。f1传入的参数flop为1，即调用了 `flop(1)` ，后面flip赋值给了一个lambda函数，这个函数传入了参数flip，返回什么暂时不知道。在f2这个lambda函数中，可以看到传入的参数为2，返回了3，因此第3个空格应该是

Python

```
flip = lambda flip: 3
```

因为lambda函数是第2个被调用的，因此最后一个空格中在flop(1)之后一定还调用了一次，并且传入的参数为2，因此最后一个空格应该是 `flop(1)(2)`

接下来f3才是最后一行调用的flip(实际上是flop)函数，传入的参数为3（因为lambda函数的返回值为3）注意，f3的返回值在 `flop` 函数定义中是flop，但是因为f3的本地变量中实际上并没有flop（变量名**不是**本地变量），因此需要到parent环境中寻找flop，而parent环境也就是Global环境中的flop已经被定义为了 `func flip` ，因此返回值是flip函数，最后f4也调用了flip

函数，传入的参数为3，但是返回值为None，说明第二个空格为None，前面的第一个空格要满足传入参数为1不满足，为3满足，可以是 `flop > 2` ，最终答案如下

```
def flip(flop):
    if flop>2:          <-- not true for flop == 1
                            true for flop == 3
        return None
    flip = lambda flip: 3
    return flip

def flop(flip):
    return flop

flip, flop = flop, flip

flip(____)(3)
      ^
      flop(1)(2)
```

2 Recursion

2.1 Recursive function

递归函数是函数体中调用了自身的函数

Python

```
def split(n):
    """Split a positive integer 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
```

递归函数中一定要有一个判断条件用来判断base case，base case中不能包含递归调用，base case是递归的终点

验证递归函数正确性的方法：

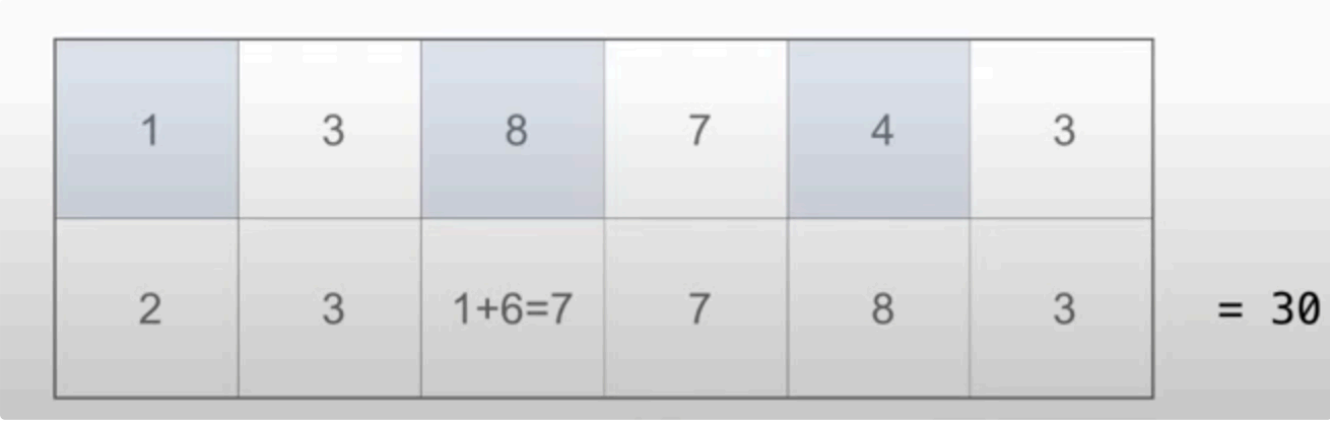
1. 验证baseline是正确的
2. 假设低一层次的函数调用 `f(n-1)` 正确，验证高一层次的函数调用 `f(n)` 是否正确

2.2 Mutual Recursion

两个函数互相递归调用

Luhn算法

从一个数的最低位开始，每2位数数字×2，当乘后的结果大于10时，将个位和十位相加，最后将所有位的数字相加



Python

```
def luhn_sum(n):
    """Return the digit sum of n computed by the Luhn algorithm.

    >>> luhn_sum(2)
    2
    >>> luhn_sum(12)
    4
    >>> luhn_sum(42)
    10
    >>> luhn_sum(138743)
    30
    >>> luhn_sum(5105105105105100) # example Mastercard
    20
    >>> luhn_sum(4012888888881881) # example Visa
    90
    >>> luhn_sum(79927398713) # from Wikipedia
    70
    """
    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):
    """Return the Luhn sum of n, doubling the last digit."""
    all_but_last, last = split(n)
    luhn_digit = sum_digits(2 * last)
    if n < 10:
        return luhn_digit
    else:
        return luhn_sum(all_but_last) + luhn_digit
```

2.3 Order of Recursive call

Inverse Cascade

Python

```
def inverse_cascade(n):
    """Print an inverse cascade of prefixes of n.

    >>> inverse_cascade(1234)
    1
    12
    123
    1234
    123
    12
    1
```

```
    """
    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)
```

2.4 Tree Recursion

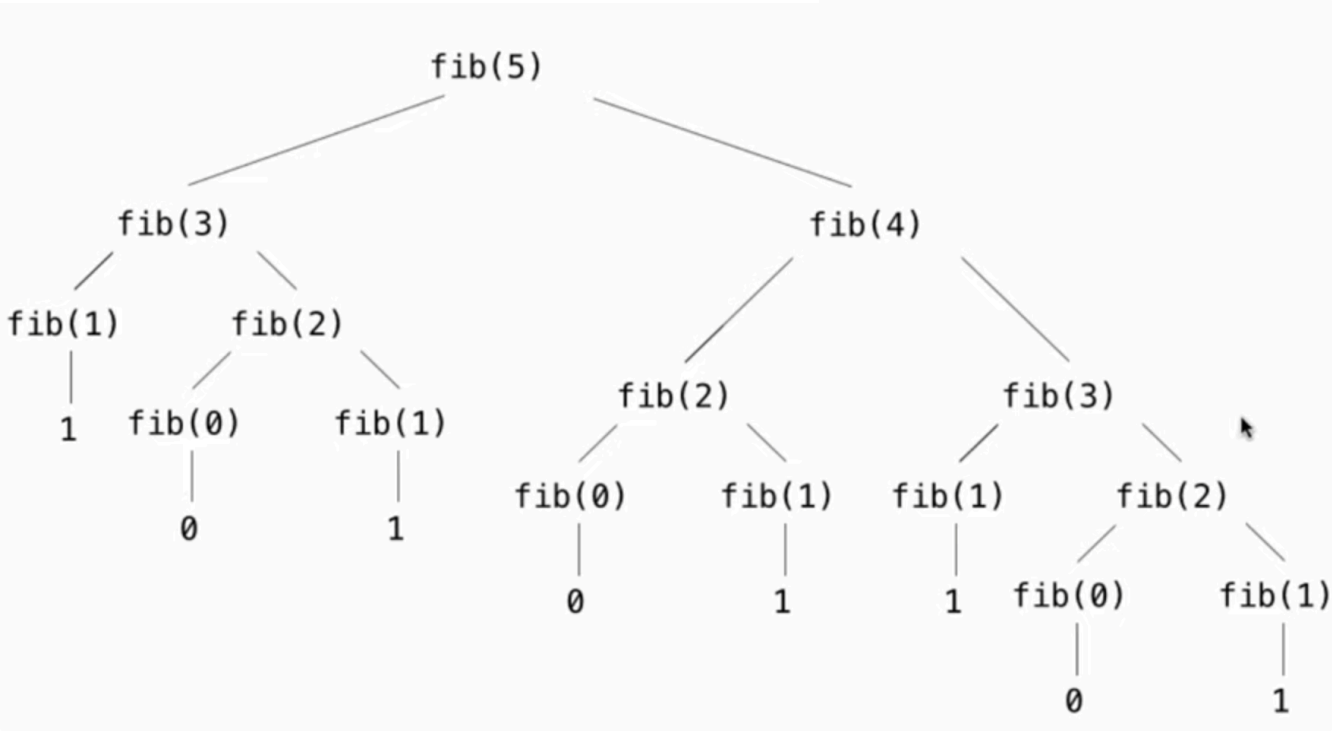
一个函数体内有超过一次对该函数自身的调用

斐波那契数列

Python

```
def fib(n):
    """Compute the nth Fibonacci number.

    >>> fib(8)
    21
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



2.5 Example

Coin split

Given a positive integer `total`, a set of coins makes change for `total` if the sum of the values of the coins is `total`. Here we will use standard US Coin values: 1, 5, 10, 25 For example, the following sets make change for `15`:

- 15 1-cent coins
- 10 1-cent, 1 5-cent coins
- 5 1-cent, 2 5-cent coins
- 5 1-cent, 1 10-cent coins
- 3 5-cent coins
- 1 5-cent, 1 10-cent coin

Thus, there are 6 ways to make change for `15`. Write a recursive function `count_coins` that takes a positive integer `total` and returns the number of ways to make change for `total` using coins. Use the `next_largest_coin` function given to you to calculate the next largest coin denomination given your current coin. I.e. `next_largest_coin(5) = 10`.

Python

```
def next_largest_coin(coin):
    """Return the next coin.
    >>> next_largest_coin(1)
    5
    >>> next_largest_coin(5)
    10
    >>> next_largest_coin(10)
    25
```

```
    >>> next_largest_coin(2) # Other values return None
    """
    if coin == 1:
        return 5
    elif coin == 5:
        return 10
    elif coin == 10:
        return 25


def count_coins(total):
    """Return the number of ways to make change for total using coins of value
    >>> count_coins(15)
    6
    >>> count_coins(10)
    4
    >>> count_coins(20)
    9
    >>> count_coins(100) # How many ways to make change for a dollar?
    242
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_coins', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
    def helper(lowest, n):
        if (lowest == None):
            return 0
        elif (lowest == n):
            return 1
        elif (lowest > n):
            return 0
        with_coin = helper(lowest, n - lowest)
        without_coin = helper(next_largest_coin(lowest), n)
        return with_coin + without_coin
    return helper(1, total)
```

Maximum Subsequence

A subsequence of a number is a series of (not necessarily contiguous) digits of the number. For example, 12345 has subsequences that include 123, 234, 124, 245, etc. Your task is to get the maximum subsequence below a certain length.

There are two key insights for this problem:

- You need to split into the cases where the ones digit is used and the one where it is not. In the case where it is, we want to reduce `t` since we used one of the digits, and in the case where it isn't we do not.

- In the case where we are using the ones digit, you need to put the digit back onto the end, and the way to attach a digit `d` to the end of a number `n` is `10 * n + d`.

Python

3 Container & Iterator

3.1 List

Python

```
>>> odds = [41, 43, 47, 49]
>>> odds[0]
41
>>> len(odds)
4
# concatenation and repetition
>>> [2, 7] + odds * 2
[2, 7, 41, 43, 47, 49, 41, 43, 47, 49]
# nested lists
>>> pairs = [[1, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

for

Python

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

首先判断是否为一个可以迭代的值，然后将和每一个这个中的元素依次绑定，并执行

range

`range` 是一个连续整数的范围，也是一个可以迭代的值，比如

Python

```
>>> list(range(-2, 2))  # list constructor
[-2, -1, 0, 1]
>>> list(range(4))
[0, 1, 2, 3]
```

list comprehension

列表推导式：`[output_expression() for(iterable value) if (conditional filter)]`

Python

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds if 25 % x == 0]
[2, 6]
```

3.2 Dictionary

```
{'key1': val1, 'key2': val2, 'key3': val3}
```

字典的key之间是没有顺序的，且不能有相同的key，且key不能是一个list

Python

```
>>> numerals = {'I':1, 'V': 5, 'X': 10}
>>> numerals['I']
1
>>> numerals.key()
dict_keys(['X', 'V', 'I'])
>>> numerals.items()
dict_items([('X', 10), ('V', 5), ('I', 1)])
>>> 'X' in numerals
True
```

3.3 Iterator

`iter(iterable)`：返回一个迭代器

`next(iterator)`：返回迭代器中指向的下一个元素

Python

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

字典也可以进行迭代

Python

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # ir iter(d)
>>> next(k)
'one'
>>> v = iter(d.values())
>>> next(v)
1
>>> i = iter(d.items())
>>> next(i)
('one', 1)
```

如果字典的大小在迭代过程中被改变，则这个迭代器**不能**继续被使用

for in iterator

Python

```
>>> r = range(3, 6)
>>> ri = iter(r)
>>> for i in ri:
...     print(i)
3
4
5
>>> for i in ri:
...     print(i)
>>>
```

迭代器在迭代一遍后再用 `for` 迭代一遍，将不会重新开始

3.4 Generator

Generator函数是一个可以 `yield` 而非 `return` 值的函数，一个正常的函数只能 `return` 一次，但是一个generator可以 `yield` 多次

Python

```
>>> def plus_minus(x):
...     yield x
...     yield -x
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
# plus_minus executed only when next is called
```

generator是一个特殊的iterator

`yield from` 可以从一个iterable中 `yield` 所有值，比如

Python

```
def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
# 与以下函数等价
def a_then_b(a, b):
    yield from a
    yield from b
```

Python

```
def countdown(k):
   if k > 0:
       yield k
       yield from countdown(k-1)
```

```
>>> t = countdown(3)
>>> next(t)
3
>>> next(t)
2
>>> next(t)
1
```

3.4 Built-in Functions for iteration

返回一个迭代器的函数：

- `map(func, iterable)`：对iterable中的x进行f(x)迭代
- `filter(func, iterable)`：在func(x)==True的情况下对iterable中的x进行迭代
- `zip(first_iter, second_iter)`：返回一个同时迭代两个可迭代对象的迭代器，next的返回对象是一个包含这两个可迭代对象元素的元组

4 Mutable Objects

mutable对象可以被改变，比如list、dictionary和set

Python

```
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits
>>> suit.pop()
'myriad'
>>> suits.remove('string')
>>> suits
['coin']
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits
['coin', 'cup', 'sword', 'club']
>>> original_suits
['coin', 'cup', 'sword', 'club']  # 2个名称指向了同一个对象，对对象的改变可以在这两个名
```

而不可变对象包括int、string、float、tuple等

Python中的变量存放的是对象引用，变量是没有类型的，可以存放任意类型的对象，因此Python实际上是一个弱类型语言。本质上，Python的变量都是指向内存对象的一个指针。不可变对象类型又叫做**值类型**，本身不能够被修改，数值的修改实际上让变量指向了一个新的对象，旧的对象被python的gc回收

Python

```
a = 1
a += 1  # a指向的对象由1这个对象变成了2这个对象
b = 2   # b此时指向的对象和a是一样的，都是2这个对象（在heap中同样数值的对象只有1个）
```

而可变类型是**引用类型**，本身允许修改

Python

```
a = [1, 2, 3]
a = [1, 2, 3]   # 这两个a指向的对象是不同的，也就是说不同的可变类型对象可以拥有同样的数值
```

当对可变类型对象进行新的赋值操作时，创建了新的对象，而进行 `.append` 这样的操作，是对原来的对象进行了修改

4.1 Tuples

元组是一组数列，**不能**改变，因此可以作为字典的键（list不能作为字典的键）

Python

```
>>> (3, 4, 5, 6)
(3, 4, 5, 6)
>>> tuple([3, 4, 5])
```

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

但是如果元组中包含了一个list，可以通过改变这个list来改变元组

Python

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 4
>>> s
([4, 2], 3 )
```

4.2 Identity Operator

`<exp0> is <exp1>` 如果 `<exp1>` 和 `<exp0>` 指向同一个对象，则返回True

4.3 Mutable default argument

如果一个函数的默认参数是一个可变对象，那么是非常危险的

Python

```
def f(s=[]):
    s.append(5)
    return len(s)
>>> f()
1
>>> f()
2
>>> f()
3
```

上述例子是因为，默认参数s=[]将其绑定到了同一个对象上，每次调用 `f()` 都会对同一个对象进行 `append`

4.4 List Operation Anatomy

Python

```
s = [2, 3]
t = [5, 6]
```

- append：

  Python

  ```
  >>> s.append(t)
  >>> t = 0
  >>> s
  ```

```
[2, 3, [5, 6]]
>>> t
0
```

注意：append的元素并不是指向t这个名称，而是被evaluate之后的那个[5, 6]对象，因此改变t之后并不会改变s

- extend

  Python

```
>>> s.extend(t)
>>> t[1] = 0
>>> s
[2, 3, 5, 6]
>>> t
[5, 0]
```

  extend是将t中的所有元素的值加到s中，也不是指向t这个名称，因此改变t之后也不会改变s

- addition & slicing

  Python

```
>>> a = s + [t]
>>> b = a[1:]
>>> a[1] = 9
>>> b[1][1] = 0
>>> s
[2, 3]
>>> t
[5, 0]
>>> a
[2, 9, [5, 0]]
>>> b
[3, [5, 0]]
```

  addition会新建一个指向原来List的对象，[t]表示的是新建一个list，这个list包含了t，因此a是一个3个元素的list，前2个是2和3，第三个指向了list t，因此改变b[1][1]会改变t

- list()

  Python

```
>>> t = list(s)
>>> s[1] = 0
>>> s
[2, 0]
>>> t
[2, 3]
```

  list是将s中的元素拷贝到一个新建的list中去，因此改变s不会改变t

4.5 Mutable Functions

Python

```
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
```

在高阶函数中定义了一个 `withdraw` 函数，这个函数中定义了一个 `nonlocal` 变量 `balance` ，当对 `balance` 进行重新赋值时，这个 `balance` 指向的是parent frame中的 `balance` 。注意， `balance` 必须被在**除了当前frame之外**的其他parent frame中被绑定

如果没有这个 `nonlocal` 声明，上述代码将会出现 `referenced before assignment` 报错，因为Python不允许修改其他frame中的变量（但是可以访问）

除了 `nonlocal` 声明之外，还可以用Mutable value来实现mutable function

Python

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw
```

这里在 `withdraw` 中，实际上并没有改变b的binding，而是改变了b这个mutable list中的一个值，从而实现了间接修改 `balance` ，由于对Python而言b指向的对象并没有改变，只是对象中的值变了，这是合法的。

5 Object-Oriented Programming

5.1 Class

类是其所有实例的模板，每个类中有很多属性和方法

Python

```
class <name>:
    <suite>
```

类构造器

类的 `__init__` 方法在用该类创建实例时被调用，第一个参数是 `self` ，用来指向这个被创建的实例，也可以加入其他的参数

Python

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

方法

方法是定义于类中的函数，使用 `obj.method()` 进行调用，传入的第一个参数是 `self` ，这个参数在调用时不需要写入

属性

可以通过 `.` 或者 `getattr` 来访问属性

Python

```
a = getattr(obj, 'attr')
```

除了实例的属性之外，类本身也可以有属性

如果一个实例没有对应的属性，但是直接对这个属性进行赋值，会在该实例上加上该属性（注意：不是在类上加属性）

5.2 Inheritance

Python

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

新的子类将从父类继承所有属性，也可以覆盖继承来的属性，在子类中重新定义原先的方法覆盖继承来的方法，但是还可以在父类中访问原来的属性或方法。

当查找一个类中的名称时，如果这个名称是在当前类中的，就返回这个类中的名称的值，否则查找父类中是否有这个名称

example

Python

```
class A:
    z = -1
    def f(self, x):
        return B(x-1)

class B(A):
    n = 4
    def __init__(self, y):
        if y:
            self.z = self.f(y)
        else:
            self.z = C(y+1)

class C(B):
    def f(self, x):
        return x

>>> a = A()
>>> b = B(1)
>>> b.n = 5
-----------
>>> C(2).n
4
>>> a.z == C.z
True
>>> a.z == b.z
False
```

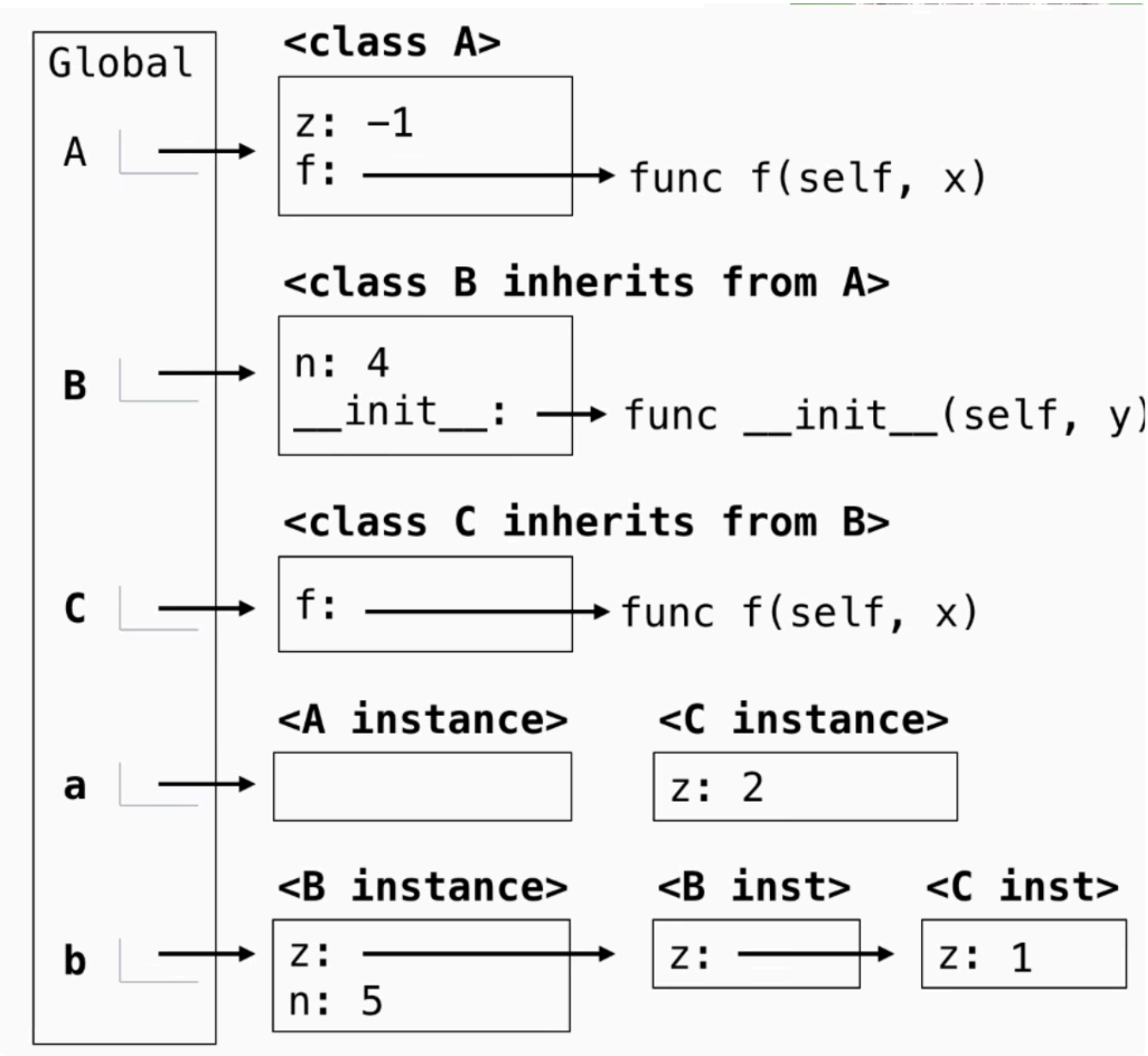
以下哪个表达式为整数?

b.z

b.z.z

b.z.z.z

b.z.z.z.z



5.3 Multiple inheritance

当一个子类继承了多个父类，就是多继承，当两个父类有相同的属性时，按照传入的父类参数从左向右查找，返回的属性是最先查找到的那个父类的属性

注意：多继承可能造成很大程度上的复杂性，因此应该注意减少多继承的使用

5.4 Representation

`repr()` 将对象转化为供解释器读取的形式，可以用 `eval` 还原对象

`obj == eval(repr(obj))`

`str()` 转化为人类可读的字符串类型

Python

```
>>> s = 'Hello'
>>> repr(s)
"'Hello'"
```

Polymorphic Function

可以对多种形式的数据进行应用的函数

Python

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

`repr()` 直接查找了class attribute中的 `__repr__` 函数而不是instance attribute

6 Data

6.1 Linked List

链表是一个pair，包括了一个值和下一个链表

`Link(3, Link(4, Link(5, Link.empty)))` , `Link.empty` 可以省略

3->4->5->empty

Python

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

Python

```
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)))
   """
   if start >= end:
       return Link.empty
   else:
       return Link(start, range_link(start + 1, end))


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)))
   """
   if s is Link.empty:
       return s
   else:
       return Link(f(s.first), map_link(f, s.rest))


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))
```

```
    """
    if s is Link.empty:
        return s
    filtered_rest = filter_link(f, s.rest)
    if f(s.first):
        return Link(s.first, filtered_rest)
    else:
        return filtered_rest
```

6.2 Tree Class

Tree是一个可以有多个 `rest` 的linked list

Python

```
class Tree:
    """A tree is a label and a list of branches."""
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(repr(self.label), branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines

    def is_leaf(self):
        return not self.branches
```

7 Scheme

Scheme是一种Lisp语言，由表达式组成，表达式包括

- 原始表达式：2， 3.3， true， quotient等
- 组合表达式：(quotient 10 2)，(not true)

括号表达式中的第一个是运算符，后面可以有0个或者多个算子

Scheme

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

7.1 Special Forms

一个不是call expression的组合式是*special form*

- **If**表达式：`(if <predicate> <consequent> <alternative>)`
- **And**和**or**表达式：`(and <e1> ... <e2>)` `(or <e1> ... <en>)`
- Binding symbol：`(define <symbol> <expression>)`
- New procedure: `(define (<symbol> <formal parameters>) <body>)`

  Scheme

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

  Scheme

  ```
  > (define (sqrt x)
      (define (update guess)
        (if (= (square guess) x)
            guess
            (update (average guess (/ x guess)))))
      (update 1))
  > (sqrt 256)
  16
  ```

- **lambda表达式**：匿名函数表达式 `(lambda (<formal-parameters>) <body>)`

以下两个表达式相同

Scheme

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

lambda表达式也可以作为call expression

Scheme

```
> ((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

- **cond**：相当于if elif else， `cond` 后面可以接多个表达式，每个表达式中有一个predicate和一个body

Scheme

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

- **begin**：可以将多个表达式组合成一个表达式

Scheme

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

- **let**：在一个表达式中暂时绑定一个值到一个symbol上

```
(let (<binding>) (<expression>))
```

Scheme

```
(define c (let ((a 3)
                (b (+ 2 2)))
                (sqrt (+ (* a a) (* b b)))))
```

这个表达式之后，a和b的值还是不知道

7.2 Scheme Lists

- `cons`：一个拥有2个参数的用于构建linked list的函数
- `car`： 返回list的第一个元素
- `cdr`： 返回list的剩余元素
- `nil`： empty list
- `list`： 新建一个list

Scheme

```
;build a list with element 2 and points to null
> (define x (cons 2 nil))
> (car x)
1
> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
> (list 1 (list 2 3) 4)
(1 (2 3) 4fil)
```

可以用 `eval` 对scheme list求值

Scheme

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

7.3 Quote and quasiquote

' 是quote标志，可以指代这个符号而不是这个符号所代表的值

Scheme

```
> (define a 4)
> 'a
a
> a
4
```

`是quasiquote标志，与quote功能类似，但是可以用,进行unquote

Scheme

```
> (define b 4)
> '(a ,(+ b 1))
(a (unquote (+ b 1)))
> `(a ,(+ b 1))
(a, 5)
```

用scheme实现while

计算从2开始的所有偶数的平方和

python中

Python

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

```
    total = total + x * x
    x = x + 2
```

scheme中

Scheme

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

8 (Optional) Tail Recursion

8.1 Functional Programming

- 所有的函数都是纯函数，返回一个数值，中间没有任何副作用
- 没有重新赋值和可变数据类型，因此没有 `for` 或者 `while` （过程中会对变量重新赋值）
- 键值绑定是永久的

优点：

- 一个表达式的值和子表达式的执行顺序无关
- 子表达式可以并行求值或者惰性求值
- *Referential transparency*：可以将任意的表达式直接替换为这个表达式的值（因为所有函数都是纯函数）

8.2 Tail calls

在python进行递归调用时，会生成很多函数栈帧，因此空间复杂度为O(n)。这些栈帧可能只是被使用了很短的一段时间，但可能会造成栈溢出，而*properly tail recursive*的语言可以实现进行尾递归调用时空间复杂度为O(1)。

为什么一定需要尾调用这个限制？因为在函数中间的调用如果没有栈帧的话，可能会对当前环境产生影响，从而影响函数中调用之后的代码正确性，因此一定要等到整个函数都执行完毕之后，在最后一个子表达式才能进行栈帧优化。

尾调用：在*tail context*中的call expression

tail context：`lambda` 表达式body中的最后一个子表达式

如果一个if表达式在tail context中，那么这个if表达式中的和也在tail context中

Scheme

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

上述例子中， `k` 和 `(factorial (- n 1) (* k n))` 都在tail context里

所有的在tail context中的 `cond` 表达式中的不是predicate的子表达式也都是尾调用

注意：如果尾调用表达式中还有一个子表达式，且这个尾调用表达式需要计算，那么这个子表达式不是尾调用（因为得到子表达式的值之后还需要进行计算，因此子表达式并不是最后一步）

Scheme

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

`(+1 (length (cdr s)))` 是尾调用，而 `(length (cdr s))` 不是，为了把它变成尾调用

Scheme

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

8.3 Tail Recursion

函数的尾调用是调用这个函数本身，这就是一个尾递归

以下面的scheme函数为例

Scheme

```
(define (sum n total)
        (if (zero? n) total
        (sum (- n 1) (+ n total))))

(sum 1001 0)
```

开启了尾递归优化的情况下：在对 `(sum 1001 0)` 进行求值时，对 `sum` 进行eval，因为sum在env中已经被定义，因此替换为前面的define的LambdaProcedure对象。然后分别对1001和0进行scheme_eval后将这两个参数scheme_apply到sum这个LambdaProcedure中，此时创建了一个新的env，绑定参数为{n:1001, total:0}，父环境为global，然后对sum函数体中的所有表达式（就只有1个）进行eval_all，而因为这个表达式的rest为nil，因此是在tail context中的，会返回一个Thunk，其环境为{n:1001, total:0}，表达式为sum的body，此时将返回到最开始的 `(sum 1001 0)` 的eval的循环中，因为返回的是一个Thunk，因此还需要对这个Thunk进行求值

对Thunk调用eval，最先解析的是if表达式，跳转到 `do_if_form` ，判断 `(zero? n)` ，发现应当跳转到alternative表达式，注意，此时的alternative表达式处在tail context中，因此不会立即进行求值跳转到新的栈帧，而是会返回一个Thunk，这个Thunk的环境还是父环境为global，表达式为 `(sum (- n 1) (+ n total))` ，再次返回到global环境中

因为返回的还是Thunk，因此需要继续对这个Thunk进行求值，对上述表达式的所有参数进行eval之后得到对sum进行apply的参数1000和1001，然后创建一个新的子环境。注意：因为此时环境还是global，因此创建的新的环境的父环境是global，而不是之前的{n:1001, total: 0}的环境，这里可以看出**Thunk的作用：将尾调用时的环境和参数保存，然后返回到global环境，再从global环境中对这个Thunk进行求值，这样可以避免直接在子环境中新创建另一个子环境，造成栈帧的递归**

9 Macro

Macro和procedure的区别在于，procedure先对其参数进行求值，然后将这些值apply到这个precedure中，而macro则直接将参数apply，然后对body进行求值，比如

Scheme

```
(define-macro (twice expr)
              (list 'begin expr expr))
(define (twice2 expr)
        (list 'begin expr expr))
> (twice (print 2))
2
2
> (twice2 (print 2))
2
(begin undefined undefined)
```

10 SQL

10.1 Declarative Programming Definition

- *declatative languages*：一个程序是对希望得到的结果的描述，怎样得到结果是由解释器负责的。申诉式语言包括SQL、Prolog等
- *imperative languages*：一个程序是对计算过程的描述，解释器负责执行这些计算过程。命令式语言包括Python、C等

SQL

```
create table cities as
    select 38 as latitude, 122 as longtitude, "Berkeley" as name union
    select 42,             71,                "Cambridge"        union
    select 45,             93,                "Minneapolis";
```

cities:

Latitude	Longtitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

10.2 SQL Intro

每个SQL语句用 ; 结尾

`select` 创建一个新的表， `create table` 给这个表一个全局名称

列描述： `select [expression] as [name]` 列描述中的 `as` 和列名称 `[name]` 都是可选的

`select` 创建一个一行的表，但是可以用 `union` 来将多个行合并为一个表格

除了创建新的表之外， `select` 还可以对已有的表进行操作，使用 `from` 来指定表。可以用 `where` 来过滤输入的表中的某些行，也可以用 `order by` 来确定这些行的排序方式

SQL

```
select [columns] from [table] where [condition] order by [order];
```

select表达式中的算数

SQL

```
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

10.3 Table

合并表格

SQL

```
-- Parents
CREATE TABLE parents AS
  SELECT "abraham" AS parent, "barack" AS child UNION
  SELECT "abraham"          , "clinton"         UNION
  SELECT "delano"           , "herbert"         UNION
  SELECT "fillmore"         , "abraham"         UNION
  SELECT "fillmore"         , "delano"          UNION
  SELECT "fillmore"         , "grover"          UNION
  SELECT "eisenhower"       , "fillmore";

-- Fur
CREATE TABLE dogs AS
  SELECT "abraham" AS name, "long" AS fur UNION
  SELECT "barack"         , "short"       UNION
  SELECT "clinton"        , "long"        UNION
  SELECT "delano"         , "long"        UNION
  SELECT "eisenhower"     , "short"       UNION
  SELECT "fillmore"       , "curly"       UNION
  SELECT "grover"         , "short"       UNION
  SELECT "herbert"        , "curly";

-- Parents of curly dogs
SELECT parent FROM parents, dogs WHERE child = name AND fur = "curly";
```

Alias and Dot Expression

当2个表格有相同的列名称时，可以用alias和 . 进行区分

筛选parents表格中拥有同一个parent的child

SQL

```
-- 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;
```

这里的 `from parents as a, parents as b` 是将2个相同的parents表格进行合并，并且给了不同的别名a和b， `a.child` 指代第一个parents表格中的child列， `b.child` 指代第二个parents表格中的child列

first	second
barack	clinton
abraham	delano
abraham	grover
delano	grover

String Expressions

`||` 字符串连接符号

SQL

```
> select "hello," || " world";
hello, world
```

SQL的index从1开始

`substr(str, start, len)` ： 从index=start开始长度为len的子字符串

SQL

```
> create table phrase as select "hello, world" as s;
> select substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1) from phrase;
low
```

更新于 2021-03-17

Others