OOP

string.append()
string.pop()
string.remove()
string.extend()
def:
"""
>>>
>>>

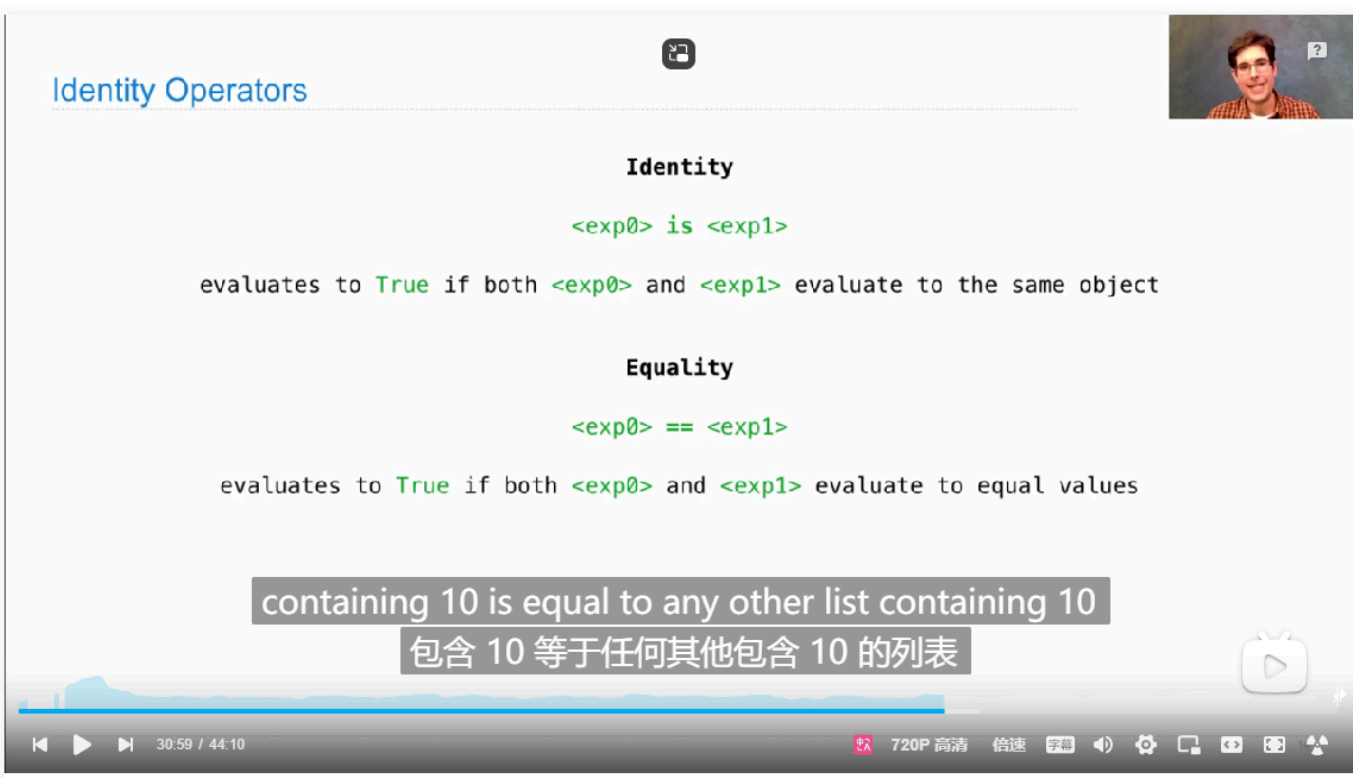
  """

unicodedata

tuple 1,2,3,4
=(1,2,3,4)

at least one thing they can agree on : the other is False
if tuple contains a list it can still change

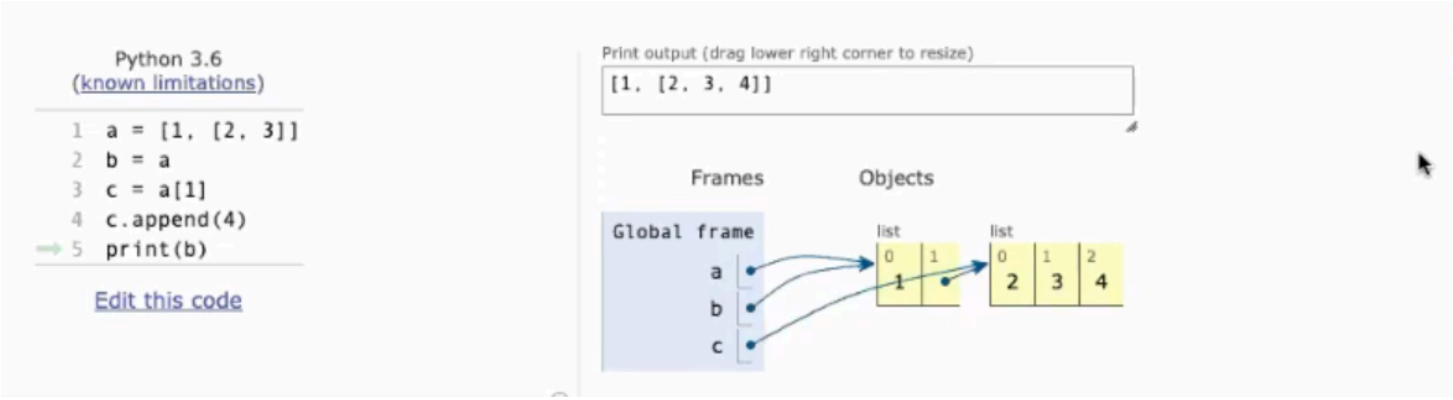
【完结 🎉】UCB CS 61A：计算机程序，2020 年秋季



notice return

notice is_

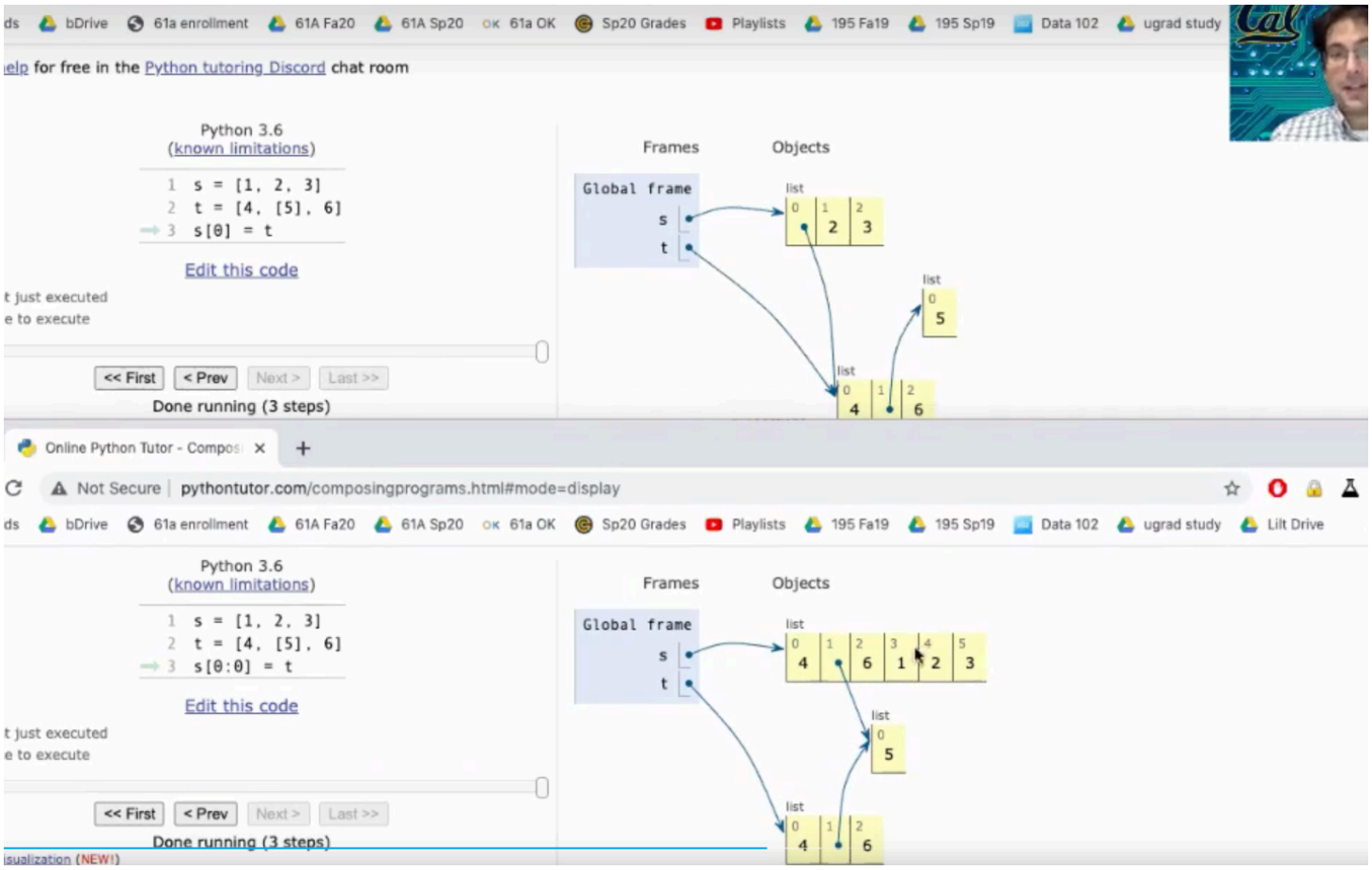
be brave to use len(list) not shamed



interesting only one copy of data

self-reference list store differs from print

selector and constructor equals bulit-in-python function
distinguish betweeen list assignment and slicing assignment



class like a blueprint
objects like houses

non-local statement can re-blind the local var to the parent frame
切记 不是没有定义该statement 只是指向两个frame 冲突
list is mutable in python can be an another way to express mutable function

distinguish between function type and is

(iter) gives a position create a iterator
(next) gives a value and change position

list shows what we have used up

dictionary.keys()
dictionary.values()
dictionary.items()

明天上完早八
静下心来
好好想代码
不再观看答案与gpt除非实在搞得太久

today's conclusion 2024/9/9

min and max
data abstraction
multiple judgement
assert statement

textbook is waiting for me to read

lazy iterartor: really funny
map
filter
zip
reversed

generator is a special kind of iterator
it's yield can stop but remember and excute many times

generator can yield from iterator

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

pop(i): Remove and return the element at index i.

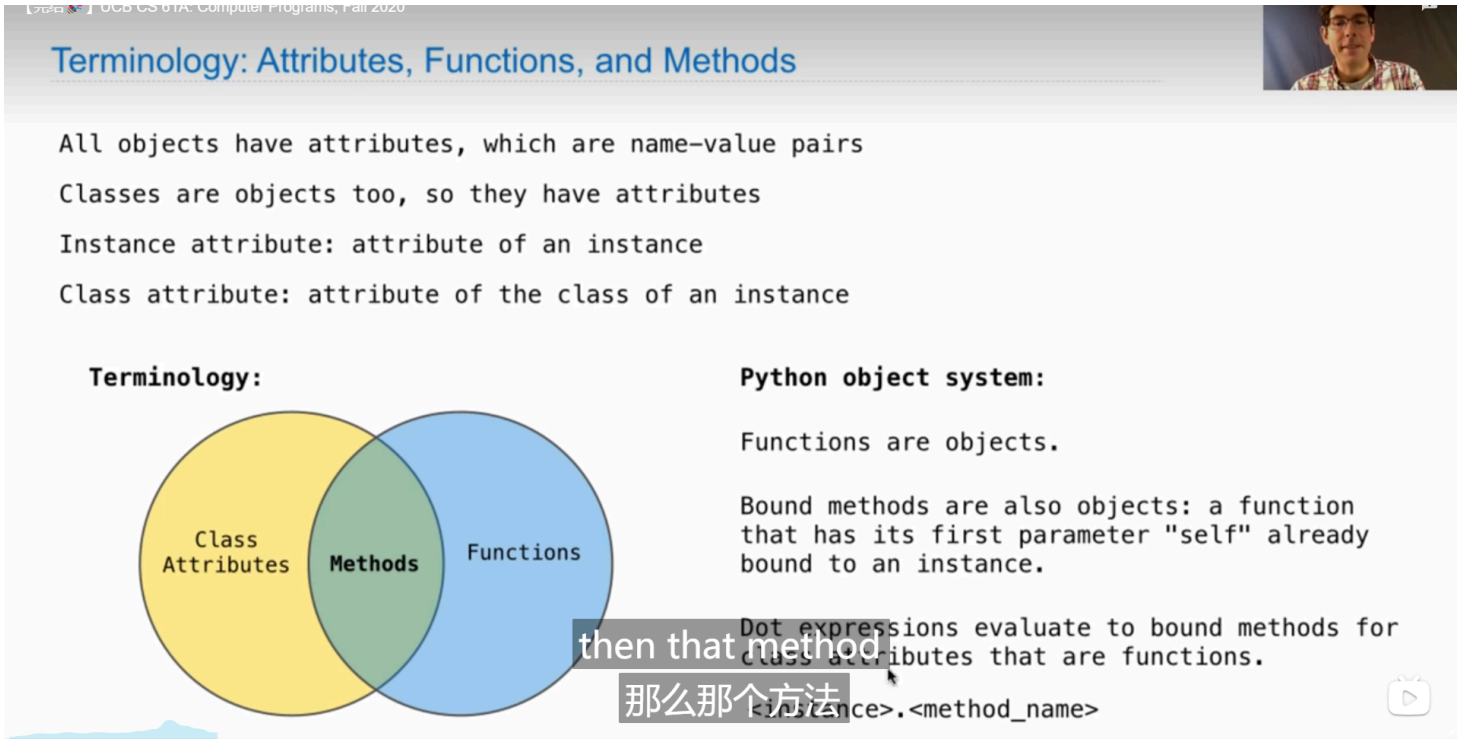
```
if not s:
    return []
elif s[0] == before:
    return [s[0]] + [after] + insert_items(s[1:], before, after)
else:
    return [s[0]] + insert_items(s[1:], before, after)
```

因生成新列表而失败
lose

反思 while 用少了 生疏了 自己也没想到 while 其实与recursive更类似，不需要for的计数存储

多看英文

while True



Attribute Assignment Statements

Account class attributes

```
interest: 0.02  0.04  0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
interest:  0.08
```

Instance attributes of tom_account

```
balance:   0
holder:    'Tom'
```

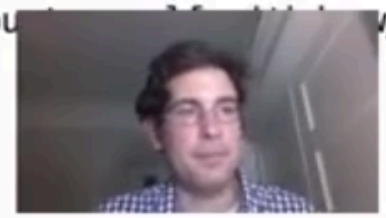
```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

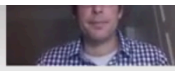
they're still there
他们还在那里

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

```
    >>> a.withdraw(90)
    'Insufficient funds'
    >>> a.balance
    10
    >>> a.interest
    0.02
    >>> Account.interest = 0.04
    >>> a.interest
    0.04
    """

    interest = 0.02  # A class attribute

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

    def deposit(self, amount):
        """Add amount to balance."""
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Subtract amount from balance."""
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self, amou                    _fee)
```



Designing for Inheritance



Don't repeat yourself; use existing implementations.

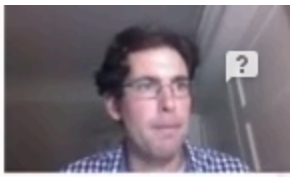
Attributes that have been overridden are still accessible via class objects.

Look up attributes on instances whenever possible.

```
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



```
class Bank:
    """A bank *has* accounts.

    >>> bank = Bank()
    >>> john = bank.open_account('John', 10)
    >>> jack = bank.open_account('Jack', 5, CheckingAccount)
    >>> john.interest
    0.02
    >>> jack.interest
    0.01
    >>> bank.pay_interest()
    >>> john.balance
    10.2
    >>> bank.too_big_to_fail()
    True
    """
    def __init__(self):
        self.accounts = []

    def open_account(self, holder, amount, kind=Account):
        account = kind(holder)
        account.deposit(amount)
        self.accounts.append(account)
        return account

    def pay_interest(self):
        for a in self.accounts:
            a.deposit(a.balance * a.interest)

    def too_big_to_fail(self):
        return len(self.accounts) > 1
```

Inheritance and Attribute Lookup

```
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
```

Which evaluates to an integer?

- b.z
- b.z.z
- ▶ b.z.z.z
- b.z.z.z.z
- None of these

and that i

iterator part:

reflection on merge:

1. next 的存储
2. 抛弃for 与 yield from 改用next
3. 条件控制 在yield的选择上的应用
4. 利用循环构建generator
5. tree abstraction 若要返回tree 往往采用tree recursive 即tree(label, f(i) for i in branches)

6. 函数加在 branches
7. 叶子往往为基本情况
8. 对于branches有的时候为遍历， 有时需要对树枝再用for, 有的时候为min max
9. 经典的构造就是if elif else 用generator递归或者用函数递归

OOP的一些重要概念：
class attribute including method == functions
look for instance attribute at first and then class attribute

argument is in **init** def make an instance it is a constructor

真他妈难

Account.deposit(a, 100)
f super without self super().withdraw(...)

tracing tree recursion is really hard but you can make it a abstraction
or use few examples
two keys:

1. don't try to trace the last call
2. think about simple samples

object and class
attribute and method

class Account:
self.interset = 0.02
def **init**(self, holder):
self.holder = holder
self.balance = 0
def deposit(self, amount):
self.balance = self.balance + amount
return self.balance
def withdraw(self, amount):
if self.balance < amount:
return 'no sufficent fund'
else:
self.balance = self.balance - amount
return self.balance

is or is not can compare independence

get attr 作为. 表达式的替代物

function 作为class attribute;
method 作为instance attribute

Account.deposite(ge_shuai, 10000000000) 2 argument

ge_shuai.deposit(10000000000000000) 1 argument

类名 capwords so account should be substitute for Account
方法名 小写

class attribute for public
class Account:
interest = 0.05

attribute
outside Account.interset = 0.07

special function
str() 供人类可读
repr() 供python解释
tue.**str**() 打印时自动调用
tue.**repr**() 交互时自动调用
repr(object) = string
eval(repr(object)) == object

eval 's argument must be string or code

repr(min) maybe different cannot generate a python expression but gives a <>

> > > repr(half)
> > > 'Fraction(1,2)'
> > > str(half)
> > > '1/2'

repr(s) give a string
and eval(repr(s)) gives the original string
str and **repr** are method
and they are polymorphic functions

str is class not a function
call str == call constructor

repr() differs from self.**repr**
repr is what we implemented

object pass message by look up attribute

an interface is a set of shared message

str() is much more complicated

format string
({0},{1}).format(x1,x2)

speial method name
bool
init
repr
str
add
float

Ratio(1,3) + Ratio(1,6)

Ratio(1,3).**add**(Ratio(1,6))

greatest common devisor

def gcd(n,m)
while n != m:
n, m = min(n,m), abs(n-m)
return n\

isinstance(instance, standard)
two important concept: type dispatching and type 强制转换

__init__后面要加引号 一直忘记

```
class Kangaroo:
def init(self):
self.content = []
def put_in_pouch(self, new):
if new in self.content:
print ("object already in pouch")
return
else:
self.content.append(new)
def str(self):
if not self.content:
print ("The kangraoo's pouch is empty")
else:
print("The kangrapp's pouch is" + str(self.content))
```

test code is very important for us to verify in the OOP courses

str is a class

isinstance 和 type 略有一些区别
repr can show anywhere as a bulit-in function
[1,2]
repr([1, 2])
[1,2].repr()

when we need to create a class, we need to def **repr**

beautiful recursive definition

```
def all_path(t):
if is_leaf(t):
yield [label(t)]
for b in branches(t):
for i in all_path(b):
yield [label(t)] + i
```

```
def print_all_path(t):
for i in all_path(t):
print(i)
```

two for loop

again repr first look at class the instance