Professor Josh Hug

Current Teaching CS61B and CS70

Will teach CS61B in this very place/time in the spring

Submit questions: pollev.com/cs61a

Attributes

Announcements

Review: Generating Partitions (from Discussion)

Generating Partitions (from Discussion)

```
def partition_gen(n, m):
    """Yield the partitions of n using parts up to size m.

    >>> for partition in sorted(partition_gen(6, 4)):
    ...     print(partition)
    1 + 1 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 3
    1 + 1 + 2 + 2
    1 + 1 + 4
    1 + 2 + 3
    2 + 2 + 2
    2 + 4
    3 + 3
    """
```

- What **small initial choice** can I make?
  - For trees, often: which branch to explore?

Use m or don’t use m

- What **recursive call for each option**?

partition_gen(n-m, m

partition_gen(n, m-1)

- How can you **combine the results** of those recursive calls?

Writing Recursive Functions (Review)

Make sure you can answer the following before you start writing code:

- What **small initial choice** can I make?
  - For trees, often: which branch to explore?

Use m or don't use m

- What **recursive call for each option**?

partition_gen(n-m, m

partition_gen(n, m-1)

- How can you **combine the results** of those recursive calls?
  - What type of values do they ~~return~~ yield?
  - What do the possible ~~return~~ yielded values mean?
  - How can you use those ~~return~~ yielded values to complete your implementation? E.g.,
    - Look to see if any option evaluated to true
    - Add up the results from each option

Choose an example! **partition_gen(6, 4)**
Write down the result of each recursive call

Method Calls

Dot Expressions

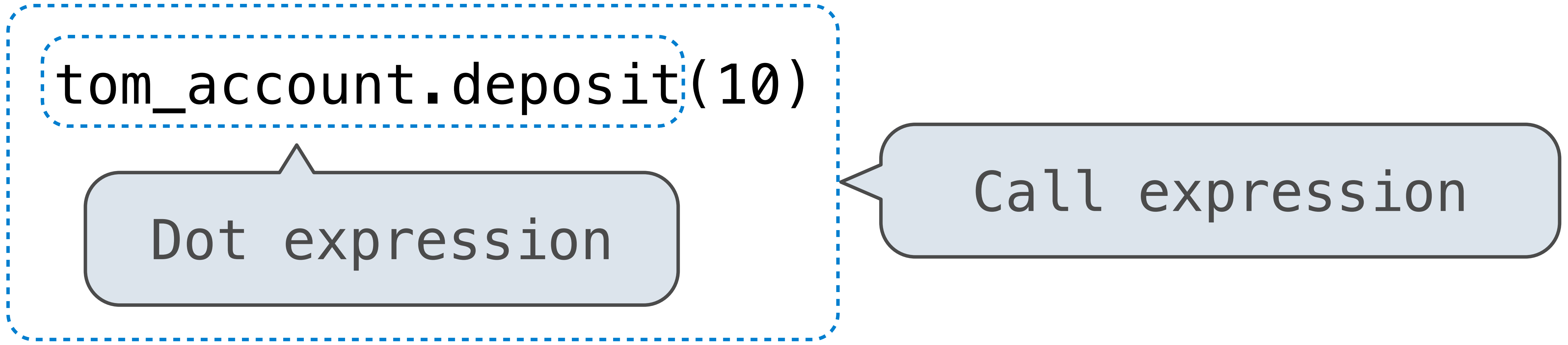
Methods are invoked using dot notation

<expression> . <name>

The <expression> can be any valid Python expression

The <name> is just a name (not a complex expression)

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>



```
tom_account.deposit(10)
```

(Demo)

Attribute Lookup

Looking Up Attributes by Name

Both instances and classes have attributes that can be looked up by dot expressions

```
<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

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Town:
   def __init__(self, w, aldo):
        if aldo == 7:
            self.street = {self.f(w): 'Waldo'}

    def f(self, x):
        return x + 1
```

```
>>> Town(1, 7).street[2]
'Waldo'
```

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Beach:
   def __init__(self):
        sand = ['Wal', 'do']
        self.dig = sand.pop

    def walk(self, x):
        self.wave = lambda y: self.dig(x) + self.dig(y)
        return self
```

Reminder: s.pop(k) removes and returns the item at index k

```
>>> Beach().walk(0).wave(0)
'Waldo'
```

Class Attributes

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

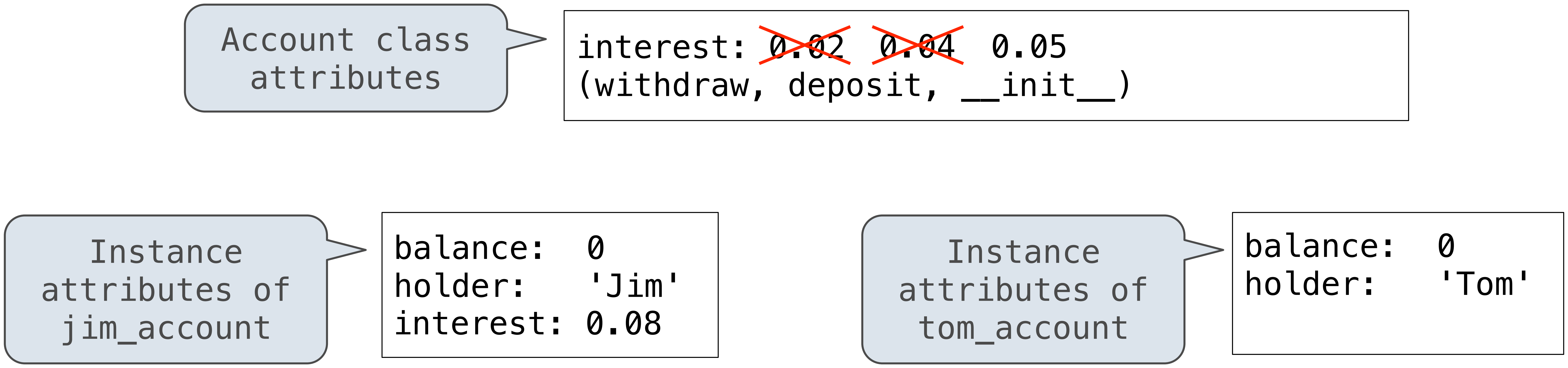
```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here
```

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

The **interest** attribute is ***not*** part of the instance; it's part of the class!

(Demo)

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
```

Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name.** Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

```
class Place:

    last = None

    def __init__(self, n):

        self.name = n

        self.then = Place.last

        Place.last = self
```

OK to write **self.last** or **type(self.last)**

Not ok to write **self.last**

```
    def print_history(self):

        print(self.name)

        if self.then is not None:

            self.then.print_history()
```

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[4].print_history()
8
6
4
2
0
>>> places[6].print_history()
12
10
8
6
4
2
0
```

More Tree Practice

Spring 2023 Midterm 2 Question 4(a)

Implement exclude, which takes a tree t and a value x. It returns a tree containing the root node of t as well as each non-root node of t with a label not equal to x. The parent of a node in the result is its nearest ancestor node that is not excluded.

```
def exclude(t, x):
    """Return a tree with the non-root nodes of tree t labeled anything but x.

    >>> t = tree(1, [tree(2, [tree(2), tree(3), tree(4)]), tree(5, [tree(1)])])
    >>> exclude(t, 2)
    [1, [3], [4], [5, [1]]]
    >>> exclude(t, 1)  # The root node cannot be excluded
    [1, [2, [2], [3], [4]], [5]]
    """
    filtered_branches = map(lambda y: _exclu
    bs = []
    for b in filtered_branches:
        if ___label(b) == x___:
            bs.___extend___(___branches(b)___)
        else:
            bs.append(b)
    return tree(label(t), bs)
```

What will the recursive call on each branch return?

What should we do with those return values?

37% of stud
got this r

30% got it right; 1 of 4 options

24% got it right

Branch has label x? Take its branches

Otherwise we're cool with the branch as-is