Object-oriented programming

Terence Parr
MSDS program
University of San Francisco

See notebook https://github.com/parrt/msds501/blob/master/notes/OO.ipynb

UNIVERSITY OF SAN FRANCISCO

The big reveal

- We've been working with functions and modules with functions, as well as defining our own functions
- It turns out, though, that we've also been working with objects all along, we just haven't recognized them as such

```
x = 'Hi'
print( x.lower() ) # send message lower to x
print( type(x) )
print( type(x.lower) )
```

```
hi
<class 'str'>
<class 'builtin_function_or_method'>
```

Classes and objects

- A *class* is the blueprint for an object and is the name of the type
- A class *encapsulates* the state (*fields*) and behavior of an object
- An object is called an *instance* of the class
- In common speech, we often use the terms interchangeably
- *Methods* are just functions associated with classes (behavior)
- Python has both functions and methods for object programming, which is why there is both **x.lower()** and **len(x)**
- **x.lower()** is implemented as **str.lower(x)**
- Objects came after the initial design in Python and the syntax is a bit awkward compared to other OO languages

Why OO programming?

- Our hunter-gatherer brain views the world as a collection of objects that interact by sending messages
- An OO programming paradigm maps well to real-world problems that we try to solve or simulate via computer
- We are at our best when programming the way our minds are hardwired to think
- OO lets us map real-world entities onto programming constructs; nouns become objects and verbs become methods

Attention: Module versus object members

- The dot '.' operator is overloaded in Python to mean both module/package member and object member access
- When we see **a.f()**, we don't know whether **f** is a member of the package identified by **a** or a method in the object referred to by **a**

```
import numpy as np
np.array([1,2,3])
```

```
array([1, 2, 3])
```

```
import math
math.log(3000)
```

```
8.00636756765025
```

Exercise

- What kind of things are the various words and subexpressions here?
  - `np.log(3)`
  - `np.linalg.norm(v)`
  - `from sklearn.ensemble import RandomForestRegressor`
  - `pd.read_csv("foo.csv")`
  - `pd.read_csv`
  - `'hi'.lower()`
  - `'hi'.lower`
  - `df_train.columns`
  - `np.pi`
  - `img = img.convert("L")`
  - `df["saledate"].dt.year`
  - `df_train.isnull().any().head(60)`

Objects group related values

- Let's baby step into the object-oriented world by associating an author and title for a few books; easiest way is with a dictionary

```
books = [
    {'title':'Gridlinked', 'author':'Neal Asher'},
    {'title':'Startide Rising', 'author':'David Brin'}
]
objviz(books)
```

Accessing “fields” with dictionary approach

- Using a small dictionary to group related values works, but has a number of significant disadvantages
  - There’s no template that ensures each dictionary has the right key and value pairs (actually Python syntax has no way to do this)
  - The notation is a bit awkward: **b[‘author’]** instead of **b.author**
  - There’s no way to associate functions with these dictionaries

```
for b in books:
    print(f"{b['author']}: {b['title']}")
```

```
Neal Asher: Gridlinked
David Brin: Startide Rising
```

Calling functions with dictionary approach

- We can obviously define a function to print out a book represented by dictionary, but there's nothing about that function that indicates it's associated with our book dictionaries

```
def show(b):
    print(f"{b['author']}: {b['title']}")

for b in books:
    show(b)
```

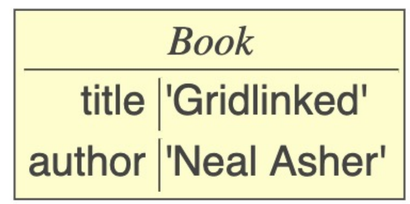
```
Neal Asher: Gridlinked
David Brin: Startide Rising
```

A basic Python class version of Book

- Compare the dictionary version to the minimal formal class version
- ("pass" just means there's nothing inside)
- We create a **Book** object/instance using the class name and parentheses
- Here, we explicitly create new fields for a **Book** object by assignment
- Notice: **b.title** vs **b['title']** notation
- There is one **Book** definition but there can be many instances

```python
class Book:
    pass

b = Book()
b.title = 'Gridlinked'
b.author = 'Neal Asher'
print(b.title, b.author)
objviz(b)
```

```
Gridlinked Neal Asher
```



Associating a function to a class

- As with fields for a specific instance, we can assign a function to the class definition using an assignment
- Then we can use OO notation **b.show()** instead of **show(b)**

```
def show(b):
    print(f"{b.author}: {b.title}")

show(b)
Book.show = show
b.show()
```

```
Neal Asher: Gridlinked
Neal Asher: Gridlinked
```

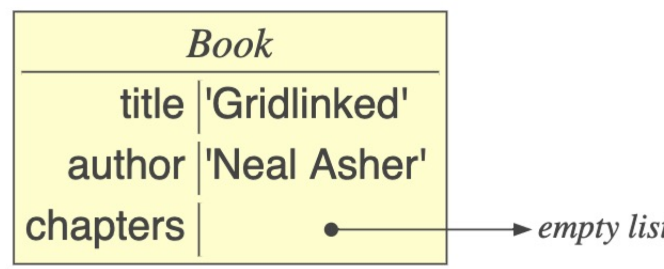
```python
b.title = 'Gridlinked'
b.author = 'Neal Asher'
```

Defining a constructor method

- Associating fields & functions to objects & classes with assignments is awkward; better to nest methods within classes
- Let's start by defining a constructor that sets initial and default field values based upon the arguments

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.chapters = []
```

```python
b = Book('Gridlinked', 'Neal Asher')
```



*Yes, **__init__** is a convention (and super weird)*

Another common method to implement

- Objects don't know how to display themselves by default:

```
print(books[0])
```

```
<__main__.Book object at 0x7f939078d6d0>
```

```
print(books[0]) # calls __str__()
```

```
Book(Gridlinked, Neal Asher)
```

- We have to define a method

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self): # called when conversion to string needed like print
        return f"Book({self.title}, {self.author})"
```

Focus on your "self"

- In methods, you must refer to fields and other methods by prefixing them with "**self.**"

```python
class Foo:
    def __init__(self):
        self.x = 0
    def foo(self):
        x = 3 # WARNING: does not alter the field! should be self.x
```

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.sold = 0 # set default

    def sell(self, n):
        self.sold += n
```

```python
b = Book('Gridlinked', 'Neal Asher')
print(b)
b.sell(100) # Book.sell(b, 100)
print(b)
```

```
Book(Gridlinked, Neal Asher, sold=0)
Book(Gridlinked, Neal Asher, sold=100)
```

Understanding methods versus functions

- **b.sell(100)** method call is translated and executed by the Python interpreter as function call **Book.sell(b,100)**
- **b** becomes parameter **self** and so the **sell()** function is updating book **b**
- Why we prefer **b.sell(100)** over **Book.sell(b,100):**
  - Instead of just functions, we send messages back and forth
  - Instead of **bark(dog)** we say **dog.bark()** or instead of **inflate(ball)** we say **ball.inflate()**

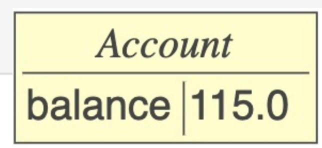
Inheritance

- Defining something new as it relates to something we already understand is usually a lot easier than starting from scratch; same is true in programming
- A *subclass* inherits from a *superclass*
- Let's start our demonstration of this by defining a simple class representing account balances

```python
class Account:
    def __init__(self, starting):
        self.balance = starting

    def add(self, value):
        self.balance += value

    def total(self):
        return self.balance
```

```python
a = Account(100.0)
a.add(15)
a.total()
```

```
115.0
```

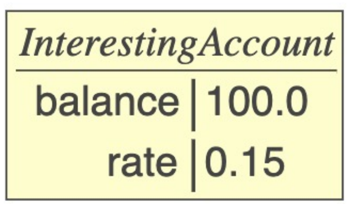


[Experiment in pythontutor]

Inheriting fields

- Inheritance behaves like an import or include operation from another class into a new class (*not exactly true*)

```python
class InterestingAccount(Account):
    def __init__(self, starting, rate):
        self.balance = starting
        self.rate = rate

b = InterestingAccount(100.0, 0.15)
```



Inheriting fields continued

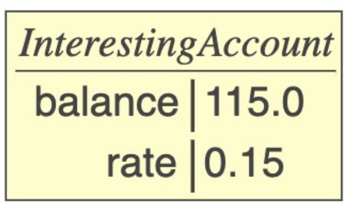
- We can also refer to the superclass constructor instead of manually assigning fields associated with the superclass; it's useful but a bit awkward
- I mention this because you will see this notation

```
class InterestingAccount(Account):
    def __init__(self, starting, rate):
        super().__init__(starting)
        self.rate = rate

b = InterestingAccount(100.0, 0.15)
```

Inheriting methods

- A class inherits all methods defined in the superclass(es) so, in this case, **InterestingAccount** inherits method **add()**

```
class InterestingAccount(Account):
    def __init__(self, starting, rate):
        super().__init__(starting)
        self.rate = rate

b = InterestingAccount(100.0, 0.15)
b.add(15)
```



[Experiment in pythontutor]

Overriding methods

- We can also override a method defined above; by defining method **total()** in the subclass, it hides the superclass definition

```
class InterestingAccount(Account):
    def __init__(self, starting, rate):
        super().__init__(starting)
        self.rate = rate
    def total(self): # OVERRIDE method
        return self.balance + self.balance * self.rate

b = InterestingAccount(100.0, 0.15)
b.add(15)
b.total()
```

```
132.25
```

We have *reused* and *refined* previous functionality

Extending functionality

- We can also *extend* the functionality by adding a method that is not in the superclass

```
class InterestingAccount(Account):
    def __init__(self, starting, rate):
        super().__init__(starting)
        self.rate = rate

    def total(self): # OVERRIDE method
        return self.balance + self.balance * self.rate

    def profit(self):
        return self.balance * self.rate
```

```
b = InterestingAccount(100.0, 0.15)
b.add(15)
b.profit()
```

```
17.25
```

[Experiment in pythontutor]