CS 61A SU26

Lecture 14: Inheritance and String Representation

July 15, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Lecture Expectations
- Computing in the News

Announcements

- **Midterm grades will be released very soon** (by tomorrow)
- Friendly reminder to submit the mid-semester feedback form, **due tonight at 11:59 pm**
  - No extensions on the feedback form
  - Worth 1 course point (graded separately from the HW assignment)
  - Help us improve the course! We read all responses, and **the more specific and actionable your feedback is, the better.**

Lecture Expectations

- Feel free to discuss with each other when we're doing a PollEv question or practice problems, but otherwise please stay quiet so that other students can hear
- Watching the lecture video playlists and/or doing the readings before lecture are **recommended but optional**
- **Lecture may repeat concepts/examples** from the videos/readings, because I do not expect prior knowledge
- The videos may cover more topics than in lecture, and vice versa. **Live lecture is the canonical scope**.
- **Lecture is intended as your "first introduction" to a concept**, so the examples may not be exam-level and the pace is intended to be slower
- **I try to release lecture materials as early as I can**, but because of other work/commitments, **I can't guarantee anything beyond releasing materials before 5 pm the day of**

Computing in the News

- Meta used AI to target workers with medical conditions for layoffs, lawsuit claims (Reuters)
- Purdue University Approves New AI Requirement For All Undergrads (Forbes)
- Brown Professor Suspects Most of His Class Used AI to Cheat (Inside Higher Ed)

Let's Design Spotify!

(reminder to self to use high contrast theme)

20 min

- Recall: Classes Brainstorm
- Implement the Classes

Recall: Classes Brainstorm

Suppose we want to create our own music streaming app, similar to Spotify.

What classes should we have?

Possible answers: User, Artist, Song, Album, Playlist

What should these classes be able to do? What are the relationships between classes?

Possible answers:

- A `User` should be able to follow an `Artist`
- A `User` should be able to add a `Song` to their Liked Songs `Playlist`
- An `Album` should be created by an `Artist` and is composed of multiple `Songs`
- A `Playlist` is composed of multiple `Songs`
- A `Song` should be created by an `Artist`

Implement the Classes

- Implement all classes
  - Some classes depend on other classes to be implemented fully before you can run the doctests
  - I recommend implementing each class in order, then running the doctests at the end
- **(Re)download** starter code `13.py` from the course website under Lecture 13
- Run doctests with `python3 -m doctest 13.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Inheritance

15 min

- Motivation
- Definitions
- Defining and Using Subclasses
- Composition
- Attribute/Method Lookup Revisited
- `type()` vs `isinstance()` Revisited

Motivation

- Objects are supposed to represent real-world things. It would be nice to have a formal way to represent special relationships between objects, e.g. categorization
- There was a lot of repeated code in the Spotify example:
  - Both `Albums` and `Playlists` can be played with the `play` method, which basically do the same thing
  - `Albums` are really just a special type of `Playlist` where the owner is an `Artist`
  - One could also argue that an `Artist` is really just a special type of `User`, since they also share attributes/behaviors (e.g. having a `name`)

Definitions

Inheritance describes an *is-a relationship* between a **subclass** (aka **child class**) and a **base class** (aka **parent class** or **superclass**).

Ex: An `Album` is a `Playlist`. A `Dog` is an `Animal`.

The subclass **inherits** (is able to access/use) the attributes and methods of its superclass.

Ex: You can `play` a `Playlist`, therefore you can also `play` an `Album`.

Defining and Using Subclasses (1 of 6)

```
class Account:
    interest = 0.02

    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
```

Defining and Using Subclasses (2 of 6)

```
class CheckingAccount(Account):
    withdraw_charge = 1
    interest = 0.01

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_charge)
```

Defining and Using Subclasses (3 of 6)

```
class CheckingAccount(Account):
    withdraw_charge = 1
    interest = 0.01

    def withdraw(self, amount):
        return super().withdraw(amount + self.withdraw_charge)
```

Defining and Using Subclasses (4 of 6)

```
>>> sriya = Account('Sriya')
>>> sriya.deposit(100)
100
>>> sriya.withdraw(50)
50
>>> emma = CheckingAccount('Emma')
>>> emma.deposit(100)
100
>>> emma.withdraw(50)
49
```

Defining and Using Subclasses (5 of 6)

```
class CheckingAccount(Account):
    withdraw_charge = 1
    interest = 0.01

    def __init__(self, holder, withdraw_charge=None):
        super().__init__(holder)
        if withdraw_charge:
            self.withdraw_charge = withdraw_charge
        else:
            self.withdraw_charge = CheckingAccount.withdraw_charge

    def withdraw(self, amount):
        return super().withdraw(amount + self.withdraw_charge)
```

Defining and Using Subclasses (6 of 6)

```
>>> emma = CheckingAccount('Emma')
>>> emma.withdraw_charge
1
>>> amy = CheckingAccount('Amy', 0.5)
>>> amy.withdraw_charge
0.5
>>> amy.deposit(100)
100
>>> amy.withdraw(50)
49.5
```

Composition (1 of 3)

Composition describes a *has-a relationship* between two classes.

Ex: A `Bank` has many `Account`s. A `Course` has many `Student`s.

Composition (2 of 3)

```
>>> bank = Bank()
>>> sriya = bank.open_account('Sriya')
>>> emma = bank.open_account('Emma', CheckingAccount)
>>> sriya.deposit(100)
100
>>> emma.deposit(100)
100
>>> bank.pay_interest()
>>> sriya.balance
102.0
>>> emma.balance
101.0
```

Composition (3 of 3)

```
class Bank:
    def __init__(self):
        self.accounts = []

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

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

Attribute/Method Lookup Revisited

To evaluate a **dot expression:**

1. Evaluate the expression to the left of the dot
2. To determine what `<name>` evaluates to:
   a. If step 1 evaluates to an instance, first check if the instance has an instance attribute/method that matches `<name>`, and if it does, return its value
   b. Otherwise, if there is no matching instance name **OR** if step 1 evaluates to a class, check if there is a class attribute* that matches `<name>`
   c. Recursively check for a name match in the superclasses (repeat steps 2a, 2b until there are no more superclasses)
   d. If there's no match, raise an `AttributeError`

type() vs isinstance() Revisited

```
>>> type(sriya)
<class 'Account'>
>>> type(emma)
<class 'CheckingAccount'>
>>> isinstance(sriya, Account)
True
>>> isinstance(emma, Account)
True
>>> isinstance(sriya, CheckingAccount)
False
>>> isinstance(emma, CheckingAccount)
True
```

String Representation

5 min

- Motivation
- `repr`
- `str`
- `__repr__` and `__str__`

Motivation

- An object should behave like the data it's representing
  - For example, by producing a string representation of itself
- Strings are important: they represent language and programs
- **In Python, all objects produce 2 string representations:**
  - `str`, a human-readable string
  - `repr`, legible to the Python interpreter
- **`str` and `repr` representations are often the same, but not always**

repr (1 of 2)

The `repr` function returns a Python expression as a string that evaluates to an equal object (by convention).

For most object types, `eval(repr(obj)) == obj`

(`eval` is a function that takes in a string and evaluates that string as a Python expression)

The result of calling `repr` on a value is what Python prints in the interpreter

```
>>> 3e5
300000.0
>>> print(repr(3e5))
300000.0
```

repr (2 of 2)

Some objects do not have a Python-readable string

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

str

The result of calling `str` on a value is what Python prints when calling the `print` function or when evaluating an f-string:

```
>>> from fractions import Fraction
>>> frac = Fraction(1, 2)
>>> repr(frac)
'Fraction(1, 2)'
>>> str(frac)
'1/2'
>>> print(frac)
1/2
>>> f'My fraction is {frac}'
'My fraction is 1/2'
```

__repr__ and __str__

The special methods `__repr__` and `__str__` are implicitly called by `repr` and `str`:

```
class Rational:
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __repr__(self):
        return f'Rational({self.numer}, {self.denom})'
    def __str__(self):
        return f'{self.numer}/{self.denom}'
>>> rat = Rational(1, 2)
>>> repr(rat)
'Rational(1, 2)'
>>> str(rat)
'1/2'
```

Practice

(reminder to self to use high contrast theme)

15 min

- Let's Redesign Spotify

Let's Redesign Spotify

- **Refactoring** code "is the process of restructuring existing source code... without changing its external behavior" (Wikipedia)
- Refactor the classes to utilize inheritance and string representation methods
  - An `Artist` is a `User`
  - An `Album` is a `Playlist`
  - Check out the new doctests
- Download starter code `14.py` from the course website under Lecture 14
- Run doctests with `python3 -m doctest 14.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?