DISCUSSION 08

OOP, Inheritance, String Representation

Mingxiao Wei
mingxiaowei@berkeley.edu

Mar 16, 2023

LOGISTICS 🏡

- Homework 06 due today 03/16
- ANTS is released! 🐜 🐛
  - Checkpoint 1 due tomorrow 03/17
  - Checkpoint 2 due next Tue 03/21
  - The whole project due next Fri 03/24
  - Submit by next Thu 03/23 for one extra point!
- Clarification: class method not in scope - you'll NOT be tested on it in assignments/exams, but you'll see it in the Ants project, in parts that are implemented already
- Come to OH! (schdule)
- Reminder - Homework 05 Recovery (Ed post #2128)

FROM LAST TIME... 👀

If you were to become an animal for the rest of your life, what would it be?

bear	pig	frog
cat	octopus	dolphin
dog	bird	cat
Cat	panda	dog
cheetah	cat	DOG
blobfish	Binturong :) They're nocturnal, live in trees, and smell like popcorn!!	Panda
A cat	dolphin	a fish
Tiger	Unicorn	Hippo
Bird	a sloth	Id be a bird
human	Cheetah or dolphin	Whale

REPRESENTATION 🎭

REPRESENTATION

- `str()` or `repr()` produces a string representation of an object
- `str(obj)`
  - returns `obj.__str__()`
  - human-readable
  - `print(obj) = print(str(obj))`
    - If `__str__` is not defined for `obj`, do `print(repr(obj))` instead
- `repr(obj)`
  - returns `obj.__repr__()`
  - computer-readable
    - By convention, this should return a string that, when evaluated, returns an object with the same value
  - `>>> obj = >>> print(repr(obj))`
- defining `__str__` and `__repr__` allows customizable string representations

STRING INTERPOLATION

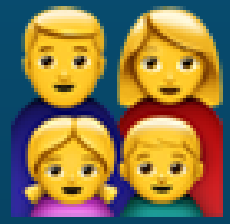
- Evaluate a string that containes some expressions
- `f'usual string and {expression}'`
  - expressions in curly braces will be evaluated and replaced with their values
  - equivalent to `'usual string and ' + str(expression)`

```
>>> x = 'cs'
>>> y = 61
>>> z = 'a'
>>> f'I love {x} {y}{z}!'
'I love cs 61a!'
>>> z = 'b'
>>> f'I love {x} {y}{z}!'
'I love cs 61b!'
```

INHERITANCE

```
class Rational:
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return f"{self.numer}/{self.denom}"
    def __repr__(self):
        return f"Rational({self.numer}, {self.denom})"
>>> a = Rational(1, 2)
>>> str(a) # return a.__str__()
'1/2'
>>> repr(a) # return a.__repr__()
'Rational(1, 2)'
>>> print(a) # equivalent to print(str(a))
1/2
>>> a # equivalent to print(repr(a))
Rational(1, 2)
```

WORKSHEET Q5, 6

INHERITANCE 

```
class Dog:
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")
class Cat:
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

INHERITANCE

- `Dog` and `Cat` have a lot in common - repeated code :(
- Solution - a base class, `Pet`, from which both classes inherit
  - DRY - Don't repeat yourself

superclass

class Pet	
is_alive	__init__(self, name, owner)
name	eat(self, thing)
owner	talk(self)

class Dog	
is_alive	__init__(self, name, owner)
name	eat(self, thing)
owner	talk(self)

class Cat	
is_alive	__init__(self, name, owner)
name	eat(self, thing)
owner	talk(self)
lives	

subclass

* super class and base class are used interchangeably

INHERITANCE

```
class Pet :  # base class
    def __init__(self, name, owner):
        self.is_alive = True      # It's alive!!!
        self.name = name
        self.owner = owner
    def eat (self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk (self):
        print(self.name)

class Dog (Pet) :  # A dog is a pet!
    def talk (self):  # overridden bc it's different from the base class
        print(self.name + ' says woof!')
```

INHERITANCE

- `class SubClass(BaseClass):`
- "is-a" relationship - a subclass is a type of base class
- By default, a subclass has the same behavior as its base class
- To make a subclass different from its base class:
  - Add attributes
    - declare additional methods/variables within the subclass
  - Override attributes
    - Class variables - reassign
    - Methods - **redefine the method with the same function signature (name and arguments)**

CALLING METHODS FROM THE BASE CLASS

When defining a method, we may want to reuse the method from the base class first, then add more to it

- `super().method(args)`
  - Can only be used inside of a class method
  - no need to pass it in `self`
- `BaseClass.method(instance, args)`
  - Can be used anywhere
  - Need to explicitly pass in the `instance`

```
class Dog(Pet):
    def __init__(self, name, owner, has_floppy_ears):
        super().__init__(name, owner)
        # alternatively, Pet.__init__(self, name, owner)
        self.has_floppy_ears = has_floppy_ears
```

WORKSHEET Q1-4

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Please leave any anonymous feedback here go.cs61a.org/mingxiao-anon
- Please do remember to fill out the form by midnight today!!