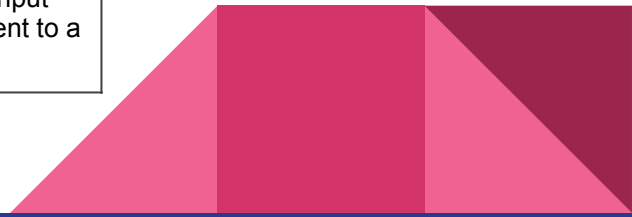
Objects

links.cs61a.org/jasonxu (password: -)

last week…

List Operations

operation	domain	range	what it does
append()	any element can be string or list	None	Adds exactly one extra element to the list; uses pointer if the input doesn't “fit”
extend()	list	None	Mutates (puts all elements from the list and adds it directly to the end of the original list)
+= ***	lst	None	Mutates
lst = lst + otherlst	lst	None	Makes new list, assigns to lst
list()	iterable	List	Iterates through the input and adds each element to a (newly-made) list



List Operations -- pt. 2

operation	domain	range	what it does
insert(i, x)	Index (if over, insert last, if under insert first) [negative indexing ok!], element	None	Adds exactly one extra element to the list at index i
remove(x)	item	None, Error	Takes out first instance of x in list, throws error otherwise

nonlocal

unbound local error

scoping -- essentially what my current frame can see, access, modify

```
def f():
        x = 5
    def g():
        print(x)
        x += 1
        return ‘success’
    return g()

def check():
        print(x)

>>>f()
Error
>>>f()
Error
```

nonlocal

unbound local error
RESOLVED

Notice placement of nonlocal

```
def f():
        x = 5
    def g():
        nonlocal x
        print(x)
        x += 1
        return ‘success’
    return g()

def check():
        print(x)

>>>f()
5
‘success’ #caution: return
value!
>>>f()
6
‘success’
```

global

explore a little!
same idea, though
exercise for the reader

```
x = 5
def f()
     global x
     print(x)
     x += 1
     return ‘success’

def check():
          print(x)

>>>f()
5
‘success’ #caution: return
value!
>>>check()
6
```

Continuing the Narrative

1. **Lists, trees**
    a. What can we do with these things?
2. **Like lego blocks**
    a. We take basic data structures and build real-world abstractions on top of them
3. **Time to create your own abstractions**
4. **ADT to the next level**

definitions

- **class**: a template for creating objects
- **instance**: a single object created from a class
- **instance attribute**: a property of an object, specific to an instance
- **class attribute**: a property of an object, shared by all instances of a class
- **method**: an action (function) that all instances of a class may perform

definitions

```
class A(<parent-class>, ...):
```

resolution order

- Resolve inheritance conflicts in order defined here

access

- Dot notation

definitions

Instantiation and Basic Access

- `example = ClassName(arg1, arg2, arg3)`
- `example.instance_var += 1 #instance-view`
- `ClassName.class_var #class-view`

Class-wide

- Modifying a class attribute from an object ***only*** affects **that** object if the value is **immutable. class var -> instance var**
- If **mutable**, all other **instances** are affected.
- **reassignment != modification**

definitions

Overriding

- Both **methods** and **values** can be overridden when there's a conflict in definition
- Can call super in child class to use parent method
- Preference given to **instance** and **current class** values/methods

definitions

- “This convention is used for **special variables or methods** (so-called “magic method”) such as __init__, __len__. These methods provide special syntactic features or do special things.”
- **__init__ is an object constructor**
  - What are the initial values of my object?
  - What do I need to know to create my object?
- **__iter__ is a required method for iterables**
- **__next__ is a method in iterators**

examples

- **__next__ is a method in iterators**
- **map**(f, *iterable*) returns a new **iterator** containing the values resulting from applying f to each value in *iterable*.
- **filter**(f, *iterable*) returns a new **iterator** containing only the values in *iterable* for which f returns True.

```
>>> dir(type([]))
['__add__', '__class__', '__contains__',
'__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append',
'clear', 'copy', 'count',
'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

definitions

***self* → refers to a specific instance of the object**

When do I pass in self manually?

- Accessing **from** class

When is self automatically passed in?

- Accessing **from** instance

Can I create a method inside a class w/o self?

- Yes/No! You can **create functions** but they should be ***instance-independent* (see foo in example)**

definitions

__repr__ → for internal representation string

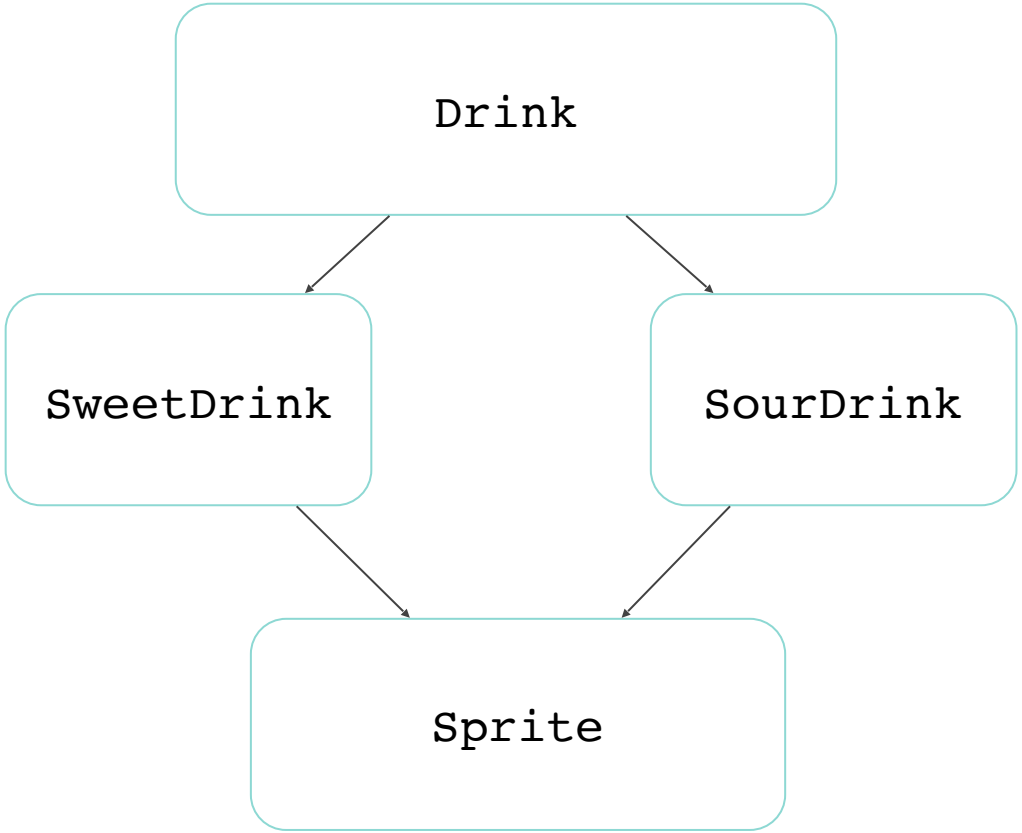
- Returns string that can recreate object (sometimes)
- **OR** “popping off the hood”
- “Car(W16 Engine, 1,103 kW (1,500 PS; 1,479 hp) at 6,700 rpm and 1,600 N·m)”

__str__ → for printing string

- What you seen when you print that object out (“what am i”)
- “2016 Bugatti Chiron, Blue”

example

inheritance
EXAMPLE



example

```
class Drink():
        size = 16
        healthy = True
        def __init__(self, my_name):
                self.name = my_name
                self.opacity = 1
```

```
class SweetDrink(Drink):
        healthy = False
        def __init__(self, my_name):
                super().__init__(my_name)
                self.opacity = 0.5

        def pour(self, amount):
                self.size -= amount
                print("sweet :)")
```

```
class SourDrink(Drink):
        def pour(self, amount):
                self.size -= amount
                print("sour")
```

```
class Sprite(SweetDrink, SourDrink):
        def __init__(self, my_name):
                self.name = my_name
                self.opacity = 0.9
        def pour_all():
        Drink.size = 0
        print("oops")
```

example

Python tutor