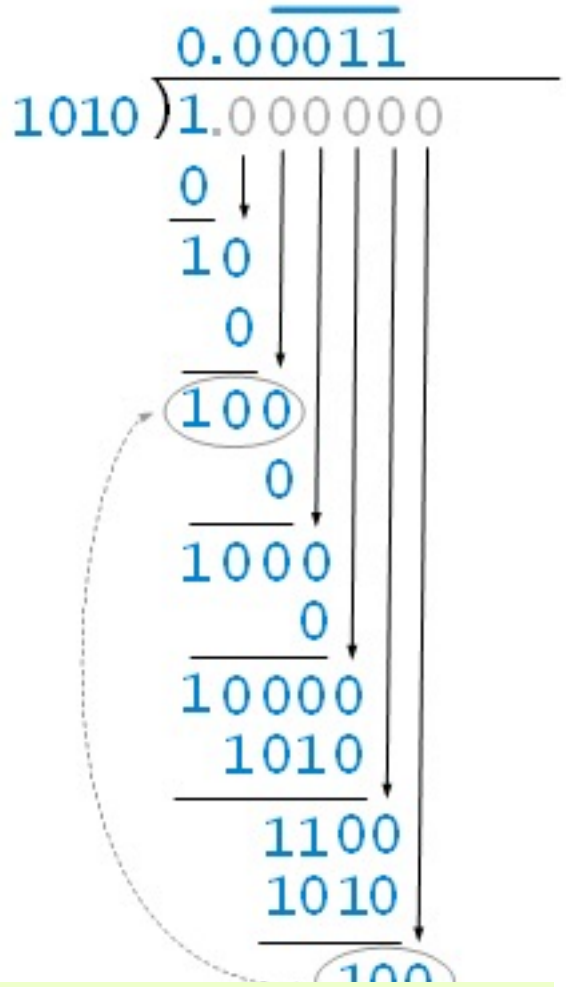
327 Object-oriented programming

Lecture 5

9/15/21

Professor Barron

Floating point money…



- Floats are rational numbers
- We know that not all rational numbers can be expressed as decimals
  - 10/3 = 3.333333333333333…
- The same problem occurs in binary but with different numbers
  - 0.1 = 0.0001100110011001…
- Rounding example
- Use Decimal class

```
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000001243449787580175325274467468261718 75')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
```

Interpreting diffs on Gradescope

- Gradescope tests will tell you what it was expecting
- Then if it has failed it will show a diff of your output and the expected output

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

- is what was expected

+ is what you had

? helps point out where the issue is

More about inheritance…

- Refresher
- Extending built-in classes
- Multiple inheritance
- More about super
- Abstract classes

Basic inheritance

- Child class inherits
  - instance variables
  - instance methods
  - this means existing relationships are included
  - only if public or protected in other languages (a newly added subclass is considered an external class)
- Hierarchical relationships
  - any shared code (data and behaviors) belongs in the parent
  - subclasses specialize
  - add new variables/methods
  - modify existing variables/methods
- What code goes in which class is all about the model

Using/overriding parent class

- super()
  - Same as super(CurrentClass, self)
- Reuse a method *specifically from a parent*
  - avoiding name clashes
  - not needed to use a parent's methods in general (already been inherited on self!)
- Often needed in __init__, but may also be used elsewhere
- May appear anywhere in a method

- If you are adding parameters then you might include those first and do this…

```
def __init__(self, new_parameter, *args, **kwargs)
    self.foo = new_parameter
    super().__init__(*args, **kwargs)
```

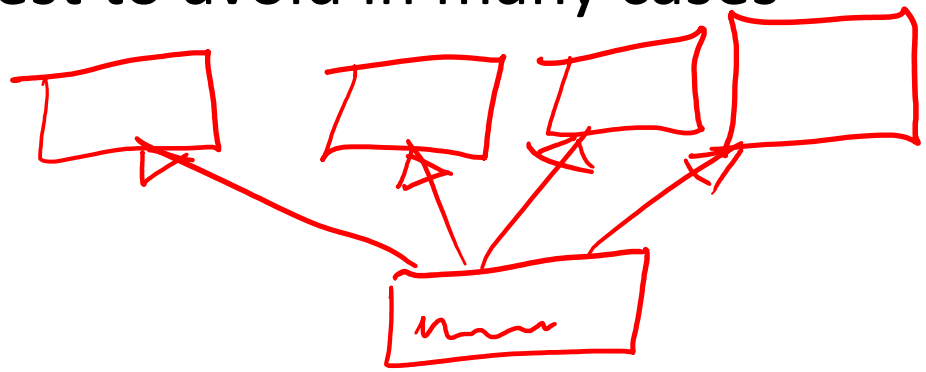
Extending built-in classes

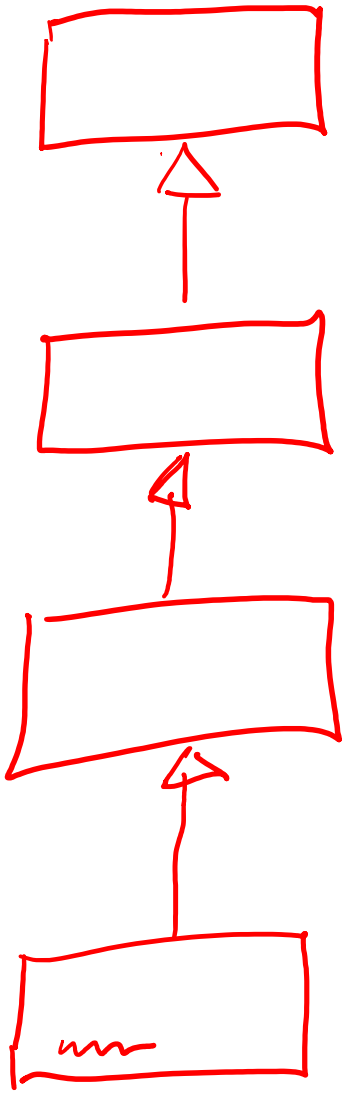
```
class ContactList(list):
    def search(self, name):
        """Return all contacts that contain the search value
        in their name."""
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

all_contacts = ContactList()
all_contacts.append(Contact("Tim", "timothy.barron@yale.edu"))
print(all_contacts[0])
```

Multiple inheritance

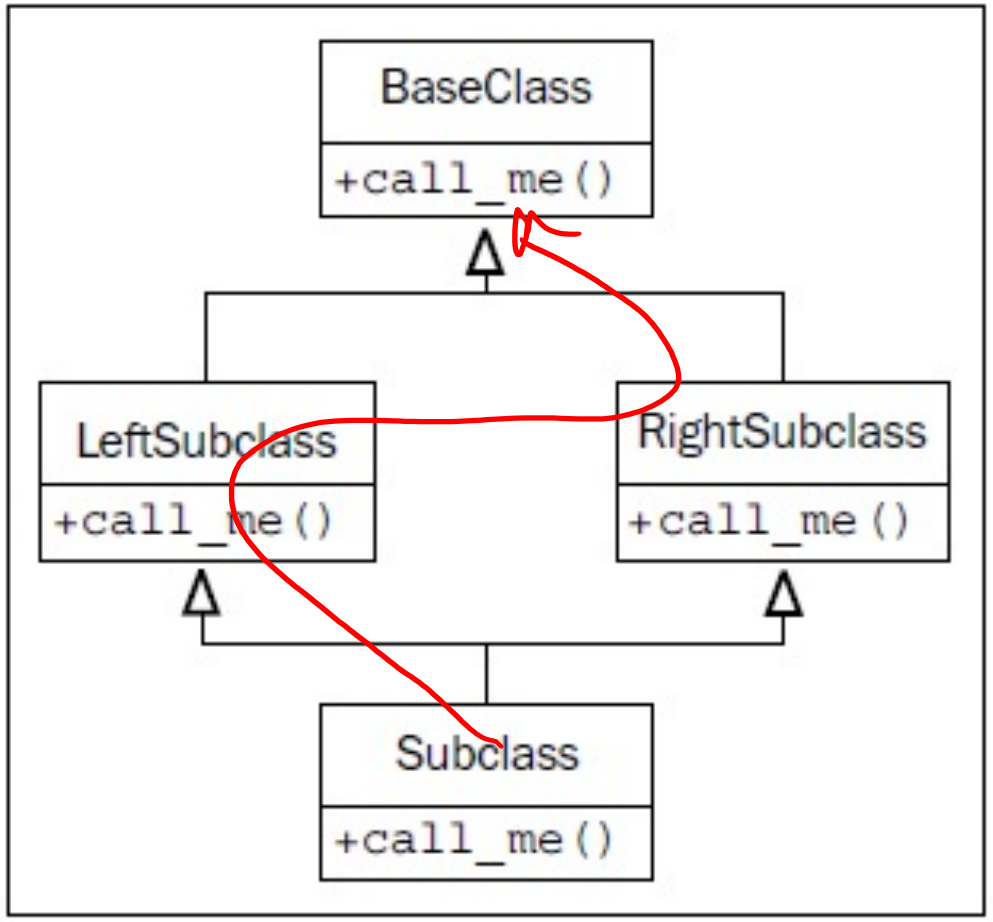
- One class inheriting from multiple others
  - class MyClass(ClassA, ClassB,…):
- Get all the attributes and behaviors of both
- What if there are conflicts?

- Super() helps out some
- Best to avoid in many cases





Diamond problem



super().call_me()

BaseClass.call_me(self)

LeftSubclass.call_me(self)
RightSubclass.call_me(self)

Mixins

- Specific use case for multiple inheritance
- Not meant to be used on their own
- Add features to multiple other classes
  - Careful about collisions
- Composition is an alternative