Chapter 03 - When Objects Are Alike

April 4, 2021

0.1 Basic Inheritance

- all python classes are subclasses of the special built-in class named `object`
- `object` provides all of the double underscore methods intended for internal use
- a `superclass` or parent class is a class that is being inherited from

- a `class variable` is part of the class definition and is shared by all instances of this class
- this means there is only one of it and we can access it as `self.<variable>`
- downside is that if you ever **set** `self.all_contacts` you will actually be creating a new instance variable associated just with the object

[7]:
```
'''
example to show that Supplier (subclass) can do what Contact (superclass) can

'''
class Contact:
    all_contacts = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

class Supplier(Contact):

    def order(self, order):
        print(
            "If this were a real system we would send "
            f"'{order}' order to '{self.name}'"
        )


c = Contact("Some Body", "somebody@email.net")
s = Supplier("Sup Piler", "supplier@example.net")
print(c.name, c.email, s.name, s.email)
```

```
Some Body somebody@email.net Sup Piler supplier@example.net
```

0.2 Extending Built-Ins

[10]:
```
class ContactList(list):
    def search(self, name):
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

c1 = Contact("John A", "johna@example.net")
c2 = Contact("John B", "johnb@example.net")
c3 = Contact("Jenna C", "johnc@example.net")

[c.name for c in Contact.all_contacts.search('John')]
```

[10]:
```
['John A', 'John B']
```

0.3 Overriding and Super

- overriding means altering or replacing a method of the superclass with a new method in the subclass

[11]:
```
class Friend(Contact):
    def __init__(self, name, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

- the problem with above is that we are writing name twice and we are forgetting to `all_contacts`
- to fix that problem we need the `super` function
- `super` call can be made inside any method

[14]:
```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

0.4 Multiple Inheritance

- if you think you need multiple inhertance you are wrong
- the simplest amd most useful form of multiple inheritance is called a `mixin`
- a `mixin` is a superclass that is not intended to exist on its own but meant to be inherited by some other class to provide extra functionality

[16]:
```
class MailSender:
    def send_mail(self, message):
        print("Sending mail to " + self.email)
        # Add e-mail logic here

class EmailableContact(Contact, MailSender):
    pass

e = EmailableContact("John Smith", "jsmith@example.net")
Contact.all_contacts
```

[16]:
```
[<__main__.Contact at 0x1bb0861f520>,
 <__main__.Contact at 0x1bb0861f4c0>,
 <__main__.Contact at 0x1bb0861f550>,
 <__main__.EmailableContact at 0x1bb085248b0>]
```

[17]:
```
e.send_mail("Hello, test e-email here")
```

```
Sending mail to jsmith@example.net
```

- we would use a standalone function over a class method because we dont have to duplicate the method for every class
- this is perferable over using meultiple inhertance

0.5 The Diamond Problem

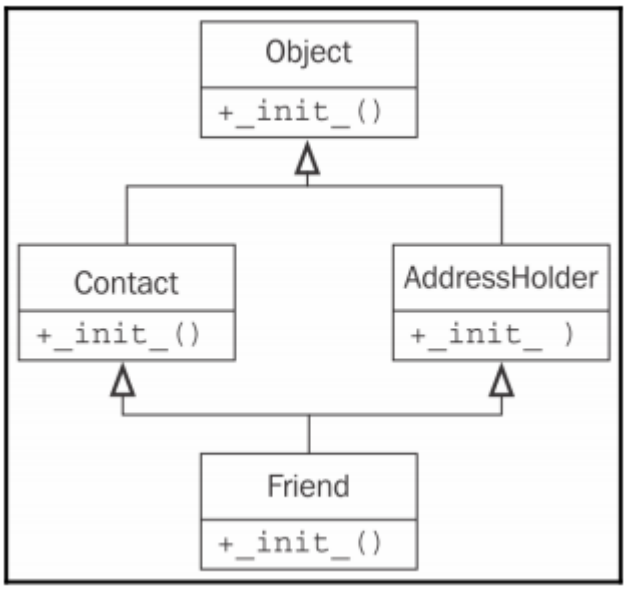
- if you inherit from two different classes, the problem is that you have two different parent `__init__` methods, both of which need to be initialized
- we could directly call the `__init__` function on each of the superclasses and explicitly pass the `self` arguments

[23]:
```
class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code


class Friend(Contact, AddressHolder):
    def __init__(
        self, name, email, phone, street, city, state, code):
```

```
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

- first problem is we could forget to initialize the superclas
- a bigger problem might be that superclass is being called multiple times becaouse of the organization of the class hierachy



- `Friend` calls `Contact` which calls `Object`
- `Friend` calls `AddressHolder` which calls `Object`
- Notice how we are calling `Object` twice?
- Base class should only be called once

0.6 Different Set of Arguments

- we dont use `super()` because we might want to have some paramaters the same and some paramater different
- we have no way of doing this
- we must aslo ensure that the method freely accepts unexpected arguments and passes them on to its `super` call

[25]:
```
class Contract:
    all_contacts = []

    def __init__(self, name="", email="", **kwargs):
        super().__init__(**kwargs)
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

```
class AddressHolder:
    def __init__(self, street="", city="", state="", code="", **kwargs):
        super().__init__(**kwargs)
        self.street = street
        self.city = city
        self.state = state
        self.code = code
class Friend(Contact, AddressHolder):
    def __init__(self, phone="", **kwargs):
        super().__init__(**kkwargs)
        self.phone = phone
```

- **kwargs basically collects any keyword arguments passed into the method that were not explicitly listed in the parameter list
- thse arguments are stored in a dictionary named kwargs
- when we call a different method (super().___init___) with a **kwargs syntax, it unpacks the dictionary and passes the results to the method as normal keyyword arguments

0.7 Polymorphism

- different behaviors happend depending on which subclass is being used, without having to explicitly know what the subclass actually is
- basically the child can take the parent method and override it

[32]:
```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"

    def play(self):
        return ("playing {} as mp3".format(self.filename))


class WavFile(AudioFile):
    ext = "wav"

    def play(self):
        return ("playing {} as wav".format(self.filename))

class OggFile(AudioFile):
    ext = "ogg"
    def play(self):
        return ("playing {} as ogg".format(self.filename))
```

```
ogg = OggFile("myfile.ogg")
print(ogg.play())
print("")

non_an_mp3 = MP3File("myfile.ogg")
```

```
playing myfile.ogg as ogg
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-32-fa4e00d20b68> in <module>
     29 print("")
     30
---> 31 non_an_mp3 = MP3File("myfile.ogg")

<ipython-input-32-fa4e00d20b68> in __init__(self, filename)
      2     def __init__(self, filename):
      3         if not filename.endswith(self.ext):
----> 4             raise Exception("Invalid file format")
      5
      6         self.filename = filename

Exception: Invalid file format
```

- polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts
- any object that supplies the correct interface can be used interchangeably in python
- often when we think we need multiple inhertance, we can just use duck typing
- duck typing is useful when dealing with mocks

0.8 Abstract Base Classes

- Abstract Base Classes define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class
- the class can extend the abstract base class itself in order to be used as an instance of that class, but it must supply all the appropriate methods

0.9 Using an Abstract Base Class

- most abstract base classes live in the `collections` module

[34]:
```
from collections.abc import Container
Container.__abstractmethods__
```

[34]:
```
frozenset({'__contains__'})
```

[38]:
```
class OddContainer:
    def __contains__(self, x):
        if not isinstance(x, int) or not x % 2:
            return False
        return True
```

[40]:
```
from collections import Container

odd_container = OddContainer()
print('isinstance(odd_container, Container)')
print(isinstance(odd_container, Container))
print("")
print('isinstance(OddContainer, Container)')
print(isinstance(OddContainer, Container))
```

```
isinstance(odd_container, Container)
True

isinstance(OddContainer, Container)
False
```

- the reson above duck typing is more awsome then classic polymorphism
- one thing cool about `Containers` **ABC** is that any class that implements it gets to use the `in` keyword for free
- `in` is just syntax sugar that delegates to the `__contains__` method

[44]:
```
print(1 in odd_container)
print(2 in odd_container)
print(3 in odd_container)
print("a string" in odd_container)
```

```
True
False
True
False
```

0.10 Creating an Abstract Base Class

- imagine we want to create something with third-party plugins
- its advisable to create an abstract base class in case to document what API the third-party plugins should provide
- the `abc` module provides the tools you need to do this

[45]:
```
import abc

class MediaLoader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def play(self):
        pass
```

```
    @abc.abstractproperty
    def ext(self):
        pass

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MediaLoader:
            attrs = set(dir(C))
            if set(cls.__abstractmethods__) <= attrs:
                return True

        return NotImplemented
```

0.11 Demystifying The Magic

- `@classmethod` decorator marks the method as a class method
- it essentially says that the method can be called on a class instead of an instantiated object

- `def __subclasshoot___(cls, C):`
- this defines the `__subclasshook__` class method