CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 5 outline

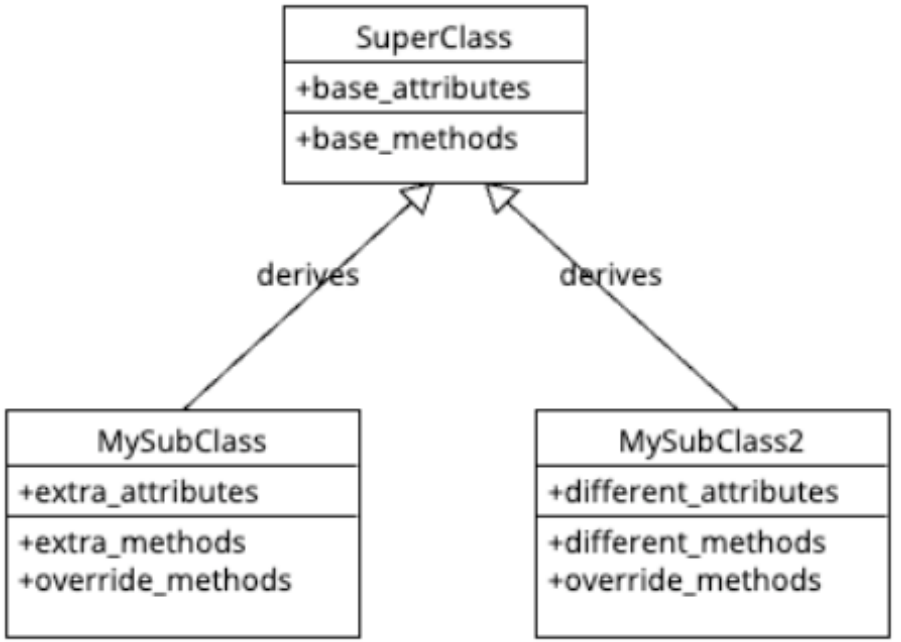
- Basic inheritance
- Inheriting from Python built-in types
- Overriding and super
- Multiple inheritance
- Polymorphism
- Duck typing
- Abstract base class
- A case study

Basic inheritance

- Inheritance allows us to define a new class (child class, sub-class) with the existing class/es (parent class/es, super-class/es)
- Technically, every class we create uses inheritance.
- Because all Python classes are subclasses of the special built-in class named ‘object’.
- ‘object’ class provides very little in terms of data and behaviors, but it allows Python to treat all objects in the same way.

Basic inheritance

- Every class will automatically inherit from ‘object’ class if it does not explicitly inherit from a different class
- A superclass (parent class) is a class that is being inherited from.
- A subclass (child class) is a class that inherits from a superclass.
- All you need to do is to include the superclass’s name inside parentheses after the subclass’s name.



Using the generalisation in UML class diagram

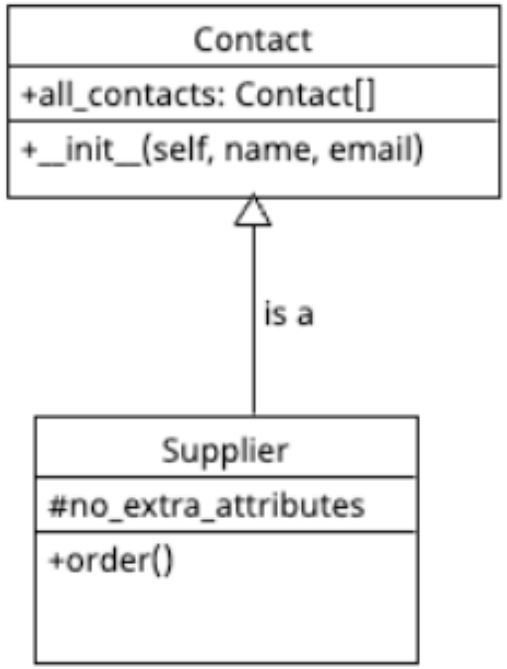
Basic inheritance

- Subclasses can access the public members (attributes and functions) in the superclasses.
- Subclasses can create new members or override existing members in the superclasses.
- Superclasses can’t access the extra members defined in the subclasses
- ‘isinstance()’ checks if an object is derived from a class
- ‘issubclass()’ checks if a class is derived from a class

```
========== RESTART: Shell ====
>>> class MySubClass(object):
...     pass
...
>>> class MySubClass2():
...     pass
...
>>> obj1=MySubClass()
>>> obj2=MySubClass2()
>>> isinstance(obj1, object)
True
>>> isinstance(obj2, object)
True
>>> issubclass(obj1,object)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    issubclass(obj1,object)
TypeError: issubclass() arg 1 must be a class
>>> issubclass(MySubClass,object)
True
>>> issubclass(MySubClass,MySubClass2)
False
```

A simple example of inheritance

- 'Contact' class is responsible for maintaining a list of all contacts.
- Each contact contains the name and the email address.
- 'Supplier' class is a subclass of 'Contact' class.
- 'order()' method is the extra method of 'Supplier' class for sending orders to the supplier (a supplier also has the name and the email address).
- We need the 'Supplier' class because we don't want to send orders to other contacts (such as family members, friends, or customers) accidently.



A simple example of inheritance

```
class Contact:
    #class veriable
    all_contacts = []
    #all_contacts = ContactList()

    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 "
            "'{}' order to '{}'".format(order, self.name)
        )
```

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net
>>> c.all_contacts
[<__main__.Contact object at 0x105d06d50>, <__main__.Supplier
object at 0x105cbff90>]
>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    c.order("I need pliers")
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' orde
r to 'Sup Plier'
```

- When we create an instance of Supplier class (subclass), the '__init__()' method in the Contact class (superclass) is called automatically.
- Instances of Contact class and Supplier class are created in the same way and are added to the same contact list, but treaded as objects of different classes.

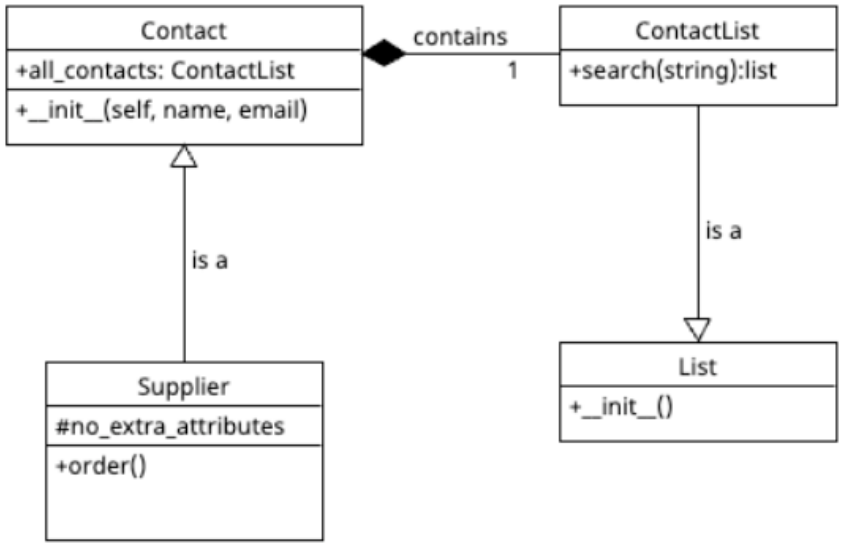
Extending Python's built-in class

- In order to search contacts in the list, we can add a 'search()' method on the Contact class.
- Actually, the 'search()' method should belong to the list, no matter it is a contact list or a shopping list.
- Python has a built-in class 'list' which can be used to create the contact list.
- If we derive the contact list from 'list' class, then we can use all built-in method of 'list' class
- We update our contact list as follows.

Extending Python's built-in class

```
1 class ContactList(list):
2     def search(self, name):
3         """Return all contacts that contain the search value
4         in their name."""
5         matching_contacts = []
6         for contact in self:
7             if name in contact.name:
8                 matching_contacts.append(contact)
9         return matching_contacts
10
11
12 class Contact:
13     #class veriable
14     #all_contacts = []
15     all_contacts = ContactList()
16
17     def __init__(self, name, email):
18         self.name = name
19         self.email = email
20         Contact.all_contacts.append(self)
```

Composition



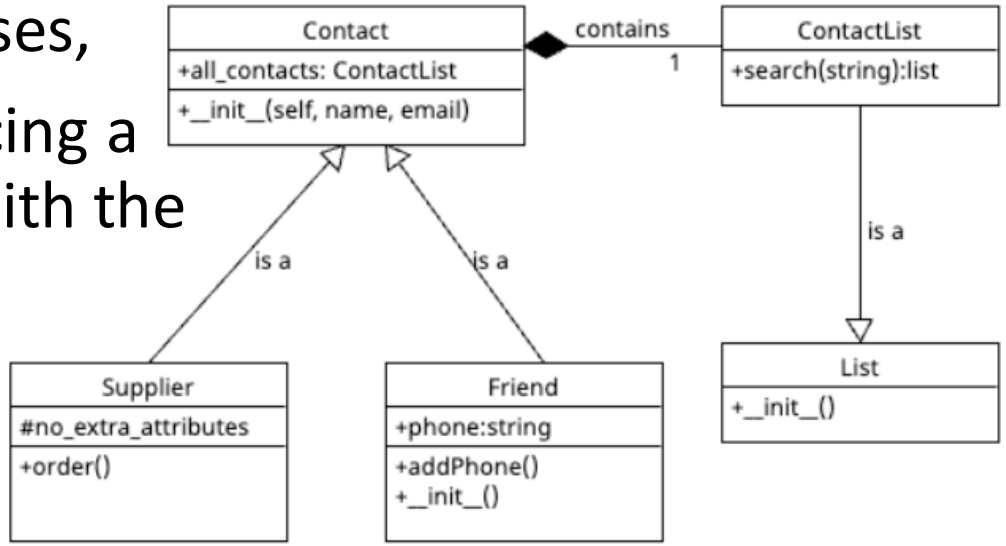
- 'ContactList' class inherits from the bulit-in 'list' class.
- 'ContactList' class can use all existing methods of 'list' class
- 'search()' method is a new method on 'ContactList' class

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@example.net")
>>> c3 = Contact("Jenna C", "jennac@example.net")
>>> isinstance(c1.all_contacts,list)
True
>>> issubclass(ContactList,list)
True
>>> c1.all_contacts
[<__main__.Contact object at 0x1050fd750>, <__main__.Contact
object at 0x105d0f7d0>, <__main__.Contact object at 0x105d0fa
50>]
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

Overriding and super

Inheritance allows us to

- add new attributes and behavior to existing classes,
- override existing methods, i.e., altering or replacing a method of the superclass with a new method (with the same name) in the subclass.



For example,

- we want to add a phone number for our close friends,
  - Solution 1: adding a phone attribute in Friend class
  - Solution 2: totally override '__init()__' method of Contact class
  - Solution 3: partially override '__init()__' method of Contact class

Overriding and super

```
31 class Friend(Contact):
32
33     #solution 1
34     def addPhone(self, phone):
35         self.phone = phone
36
37
38     #solution 2
39     def __init__(self, name, email, phone):
40         self.name = name
41         self.email = email
42         self.phone = phone
43         Contact.all_contacts.append(self)
44
45     #solution 3
46     def __init__(self, name, email, phone):
47         super().__init__(name,email)
48         self.phone = phone
```

Please comment other solutions when you use one

Solution 1:

- does not use inheritance.
- the 'phone' attribute can't be used by its subclass

Solution 2:

- use inheritance.
- manually add all existing attributes and itself to the all_contact list
- hard to maintain because of duplicate codes in the super and sub classes (duplicate codes are evil in OOP ☹)

Solution 3:

- use inheritance.
- use 'super()' method to automatically initialise the existing attributes
- automatically add itself to the all_contact list
- manually add the **extra attributes** only
- no duplicate code and easy to maintain ☺

Overriding and super

```
class Friend(Contact):

    #solution 1
    def addPhone(self, phone):
        self.phone = phone


f1= Friend("Friend A", "frienda@example.net")
f1.addPhone("1234567")
print(f1.name, f1.email, f1.phone)
Friend A frienda@example.net 1234567
```

```
class Friend(Contact):

    #solution 2
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
        Contact.all_contacts.append(self)


f2=Friend("Friend B", "friendb@example.net", "1111111")
print(f2.name, f2.email, f2.phone)
Friend B friendb@example.net 1111111
```

```
class Friend(Contact):

    #solution 3
    def __init__(self, name, email, phone):
        super().__init__(name,email)
        self.phone = phone


f3=Friend("Friend C", "friendc@example.net", "2222222")
print(f3.name, f3.email, f3.phone)
Friend C friendc@example.net 2222222
```

Which solution is the best?

Overriding and super

Solution 3 is the best: what happens after we call ‘super()’ method?

1. Gets the instance of the parent object using keyword ‘super’, i.e., ‘super = SuperClass(name, email)’;
2. Call ‘__init()__’ method of the superclass with the parent object, i.e., super.__init(name, email)__;
3. Initalise its own attribute, i.e., phone attribute;
4. Return the instance of the child object with the initialized values.

Note:

- ‘super()’ method can be made inside any methods.
- If call super() firstly. The superclass method will be called first then execute the subclass method codes.
- All method can be modified via overriding and calls to super().
- The call to super() can also be made at any point in the method. For example, we may need to manipulate or validate incoming parameters before forwarding them to the superclass.

Multiple inheritance

- Python supports multiple inheritance.
- A subclass that inherits from more than one parent class is able to access functionality from all of them.
- In practice, less useful than it sounds.
- Many expert programmers recommend against using it.
- Java does not support multiple inheritance, but uses another technique named ‘interface’ to resolve the problem of using methods from different superclasses.
- We will find out how the multiple inheritance works and why it is not recommended.

Mixin

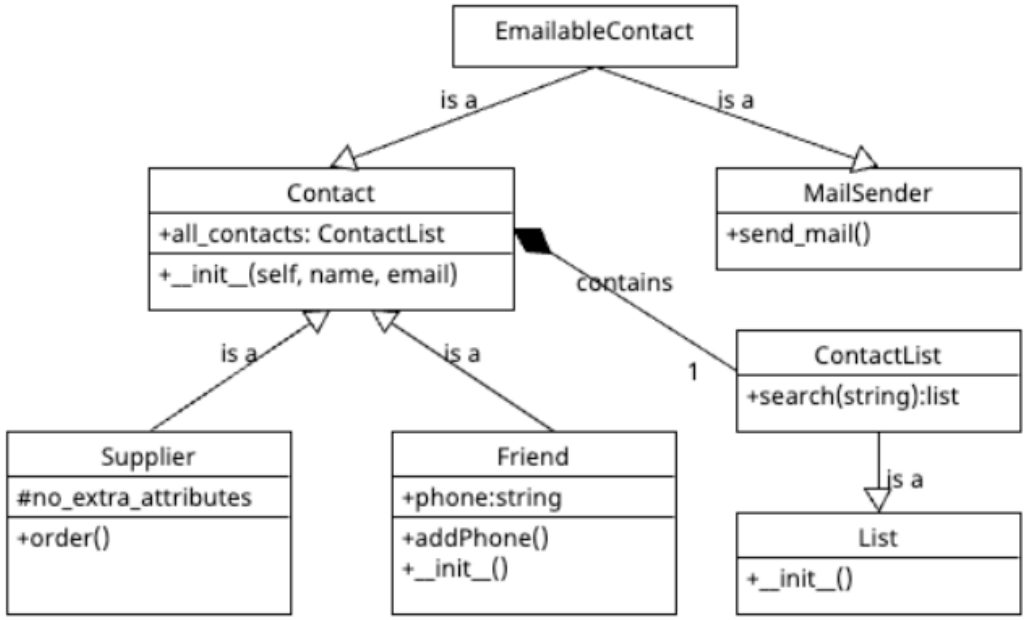
- Mixin is the simplest and most useful form of multiple inheritance.
- A mixin is a superclass that is meant to be inherited by some other class to provide extra functionality.
- Mixins are not supposed to work alone.

For example, we want to add functionality to our ‘Contact’ class that allows sending an email. Because sending email is a common task that we might want to use on other classes. So, we write a simple mixin class to do the emailing for us.

Mixin

- We define a mixin class, i.e., MailSender class
- MailSender class has a function 'send_mail(message)'
- Multiple inheritance is used to create a new subclass 'EmailableContact'.
- 'EmailableContact' class has two parent classes, i.e., Contact and MailSender
- Instances of EmailableContact class can access public members from both parent classes, i.e., sending an email to some contacts.

```
>>> 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")
>>> e.send_mail("Hello, test email here")
Sending mail to jsmith@example.net
>>> issubclass(EmailableContact, Contact)
True
>>> issubclass(EmailableContact, MailSender)
True
```



Potential issues with multiple inheritance

- Multiple inheritance works all right when mixing methods from different classes.
- But it gets very messy when we call methods on the superclass.
- There are multiple superclasses.
- How do we know which one to call?
- How do we know what order to call them in?

For example, we want to add a home address to the ‘Friend’ class.

- Solution 1 (attribute): define a new ‘address’ attribute on the ‘Friend’ class to store the home address.
- Solution 2 (composition): create a new ‘Address’ class to hold the home address, and then pass an instance of this class to the ‘Friend’ class.
- Solution 3 (multi-inheritance): create an ‘AddressHolder’ class and use it as the second parent class of the ‘Friend’ class

Which solution is the best?

The diamond problem

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

Because ‘Friend’ class has two superclasses and both of them need to be initialized. How do we do this? We could start with a naïve approach, i.e., calling their ‘__init__’ methods manually

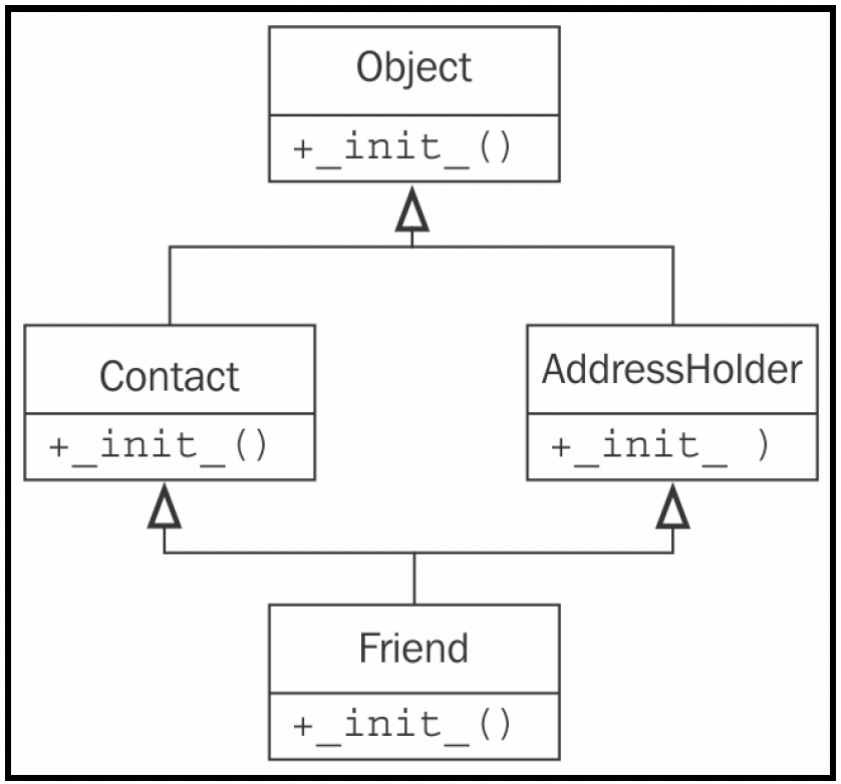
```
class Friend(Contact, AddressHolder):
    def __init__(
        self, name, email, phone, street, city, state, code):
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

Potential issues:

1. A superclass may go uninitialised if we neglect to explicitly call the initializer.
2. A superclass may be called multiple times because of the organization of the class hierarchy.

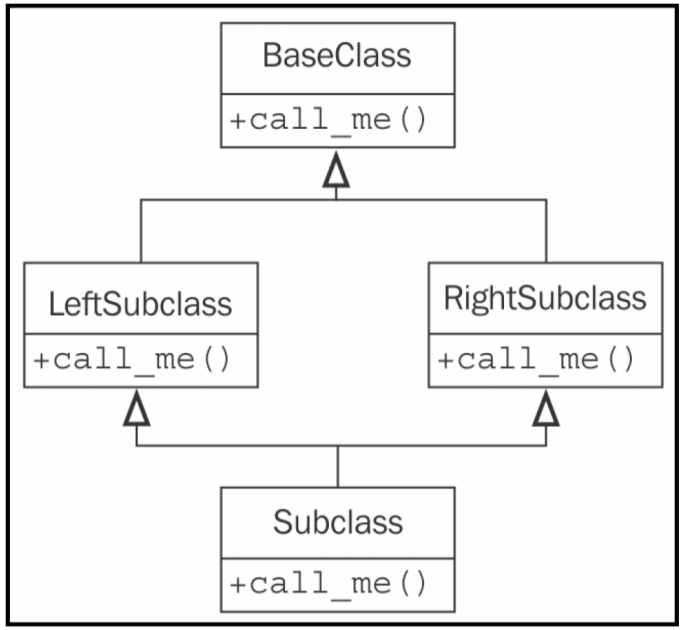
The diamond problem

- The __init__ method from the Friend class first calls __init__ on Contact, which implicitly initializes the object superclass.
- Then Friend calls __init__ on AddressHolder which also initializes the object superclass again.
- The parent class has been set up **Twice** !!!
- It could be a disaster in some situations, such as connecting to a database twice for every request.



The diamond problem

```
1 class BaseClass:
2     num_base_calls = 0
3
4     def call_me(self):
5         print("Calling method on Base Class")
6         self.num_base_calls += 1
7
8
9 class LeftSubclass(BaseClass):
10    num_left_calls = 0
11
12    def call_me(self):
13        super().call_me()
14        print("Calling method on Left Subclass")
15        self.num_left_calls += 1
16
17
18 class RightSubclass(BaseClass):
19    num_right_calls = 0
20
21    def call_me(self):
22        super().call_me()
23        print("Calling method on Right Subclass")
24        self.num_right_calls += 1
25
26
27 class Subclass(LeftSubclass, RightSubclass):
28    num_sub_calls = 0
29
30    def call_me(self):
31        #super().call_me()
32        LeftSubclass.call_me(self)
33        RightSubclass.call_me(self)
34        print("Calling method on Subclass")
35        self.num_sub_calls += 1
```



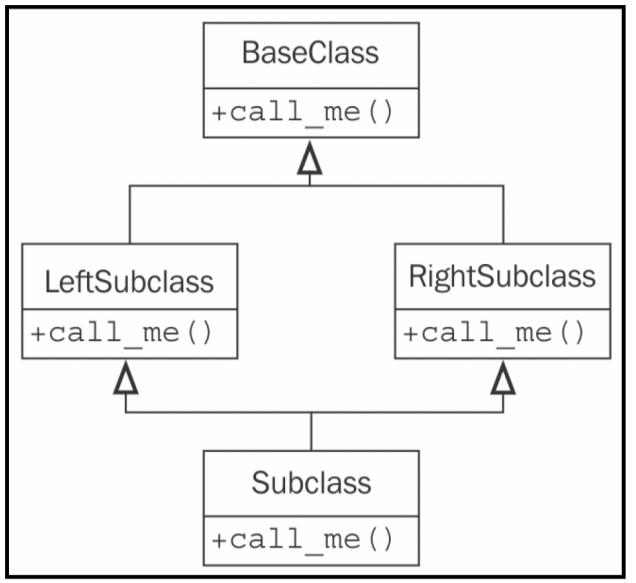

```
>>> s = Subclass()
>>> s.call_me()
    Calling method on Base Class
    Calling method on Right Subclass
    Calling method on Left Subclass
    Calling method on Base Class
    Calling method on Right Subclass
    Calling method on Subclass
>>> print(
...     s.num_sub_calls,
...     s.num_left_calls,
...     s.num_right_calls,
...     s.num_base_calls)
1 1 2 2
```

The base class's 'call_me' method being called twice.

Think about what if this happens on your internet banking when you try to withdraw…

The diamond problem solution: super()

```
1 class BaseClass:
2     num_base_calls = 0
3
4     def call_me(self):
5         print("Calling method on Base Class")
6         self.num_base_calls += 1
7
8
9 class LeftSubclass(BaseClass):
10    num_left_calls = 0
11
12    def call_me(self):
13        super().call_me()
14        print("Calling method on Left Subclass")
15        self.num_left_calls += 1
16
17
18 class RightSubclass(BaseClass):
19    num_right_calls = 0
20
21    def call_me(self):
22        super().call_me()
23        print("Calling method on Right Subclass")
24        self.num_right_calls += 1
25
26
27 class Subclass(LeftSubclass, RightSubclass):
28    num_sub_calls = 0
29
30    def call_me(self):
31        super().call_me()
32        print("Calling method on Subclass")
33        self.num_sub_calls += 1
```




```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
>>> print(
...     s.num_sub_calls,
...     s.num_left_calls,
...     s.num_right_calls,
...     s.num_base_calls)
1 1 1 1
```

'super' method saves us again this time.

'super' call is not calling the method on the superclass of LeftSubclass, but calling RightSubclass.

Different sets of arguments

- The situation is more complicated in our ‘Friend’ multiple inheritance example.

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

- How to manage different sets of arguments when using super?
- How can we pass the extra arguments to super to ensure the subsequent calls on other subclasses can receive the right arguments?
- For example, if the first call to super passes the name and email arguments to Contact.__init__, and Contact.__init__ then calls super. It needs to be able to pass the address-related arguments to the next method, which is AddressHolder.__init__.

Different sets of arguments

- Sadly, the only way to solve this problem is to plan from the beginning.
- We have to design our base class parameter list to accept keyword arguments from any parameters that are not required by every subclass implementation.
- We must ensure that method freely accepts unexpected arguments and passes them on to its super class, in case they are necessary to later methods in the inheritance order.
- Python’s function parameter syntax provides all the tools we need to do this, but it makes the overall code look cumbersome。

*arg and **kwargs in Python

Python has two special symbols for passing arguments:

- *args (Non-Keyword Arguments): taking in a variable number of arguments. Any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments)
- **kwargs (Keyword Arguments): using to pass a keyworded, variable-length argument list. It is a dictionary that maps each keyword to the value that we pass alongside it.

*argv and **kwargs in Python

Python3

```
def myFun(arg1, *argv):
    print("First argument :", arg1)
    for arg in argv:
        print("Next argument through *argv :", arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : GeeksforGeeks
```

Python

```
# defining car class
class car():
    # args receives unlimited no. of arguments as an array
    def __init__(self, *args):
        # access args index like array does
        self.speed = args[0]
        self.color = args[1]


# creating objects of car class
audi = car(200, 'red')
bmw = car(250, 'black')
mb = car(190, 'white')

# printing the color and speed of the cars
print(audi.color)
print(bmw.speed)
```

Output:

```
red
250
```

*arg and **kwargs in Python

Python3

```python
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Output:

```
first == Geeks
mid == for
last == Geeks
```

Python

```python
# defining car class
class car():
    # args receives unlimited no. of arguments as an array
    def __init__(self, **kwargs):
        # access args index like array does
        self.speed = kwargs['s']
        self.color = kwargs['c']


# creating objects of car class
audi = car(s=200, c='red')
bmw = car(s=250, c='black')
mb = car(s=190, c='white')

# printing the color and speed of cars
print(audi.color)
print(bmw.speed)
```

Output:

```
red
250
```

Different sets of arguments

Here is a 'proper' version of the Friend multiple inheritance code.

```
1 class Contact:
2     all_contacts = []
3
4     def __init__(self, name="", email="", **kwargs):
5         super().__init__(**kwargs)
6         self.name = name
7         self.email = email
8         self.all_contacts.append(self)
9
10
11 class AddressHolder:
12     def __init__(self, street, city, state, code):
13         self.street = street
14         self.city = city
15         self.state = state
16         self.code = code
17
18
19 class Friend(Contact, AddressHolder):
20     def __init__(self, phone="", **kwargs):
21         super().__init__(**kwargs)
22         self.phone = phone
```

- We changed all arguments to keyword arguments by giving them an empty string as a default value.
- We also ensure a '**kwargs' parameter is included to capture any additional parameters that our particular method doesn't know what to do with. It passes these parameters up to the next class with the super call.
- **kwargs collects any keyword arguments passed into the method that were not explicitly listed in the parameter list.
- These arguments are stored in a dictionary. We can call the dictionary whatever we like, but convention suggests **kwargs.
- When we call a different method with a **kwargs syntax, it unpacks the dictionary and passes the results to the method as normal keyword arguments.

More discussions on the multiple inheritance

- Actually, the above implementation is still insufficient if we want to reuse variables in parent classes.
- When we pass the **kwargs variable to super, the dictionary may not include any of the variables that were included as explicit keyword arguments.
- For example, in Friend.__init__, the call to super does not have phone in the kwargs dictionary. If any of the other classes need the phone parameter, we need to ensure it is in the dictionary that was passed. Worse, if we forget to do this, it will be extremely frustrating to debug because the superclass will not complain, but will simply assign the **default value** (in this case, an empty string) to the variable.

Potential solutions:

- Don't include phone as an explicit keyword argument. Instead, leave it in the **kwargs dictionary. Friend can look it up using the kwargs['phone'] syntax. When it passes **kwargs to the super call, phone will still be in the dictionary.
- Make phone an explicit keyword argument, but update the kwargs dictionary before passing it to super, using the standard dictionary kwargs['phone'] = phone syntax.
- Make phone an explicit keyword argument, but update the kwargs dictionary using the kwargs.update method. This is useful if you have several arguments to update.
- Make phone an explicit keyword argument, but pass it to the super call explicitly with the super().__init__(phone=phone, **kwargs) syntax.

A even better solution: Do not use multiple inheritance. Try using composition or other design patterns (will introduce in later lectures)

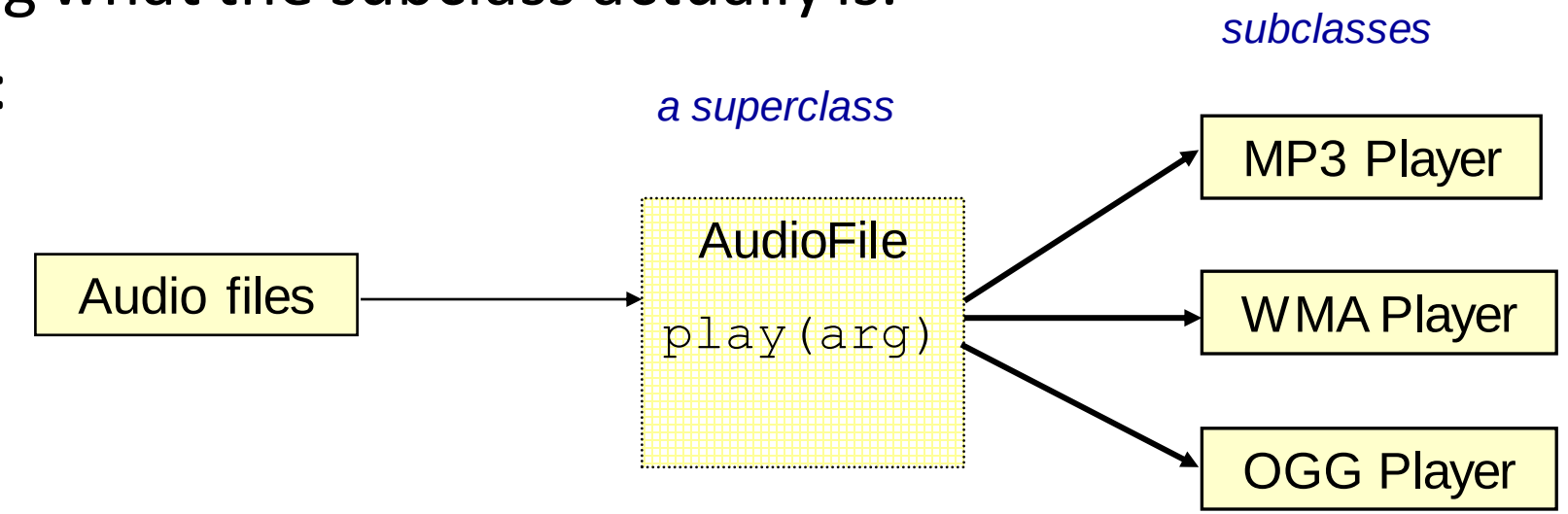
Alternative solutions besides mixin

- We could have used single inheritance and added the send_mail function to the subclass. The disadvantage here is that the email functionality then has to be duplicated for any other classes that need an email.
- We can create a standalone Python function for sending an email, and just call that function with the correct email address supplied as a parameter when the email needs to be sent.
- We could have explored a few ways of using composition instead of inheritance.
- For example, EmailableContact could have a MailSender object as a property instead of inheriting from it.

Polymorphism

- Polymorphism refers to a simple concept: different behaviours happen depending on which subclass is being used without explicitly knowing what the subclass actually is.

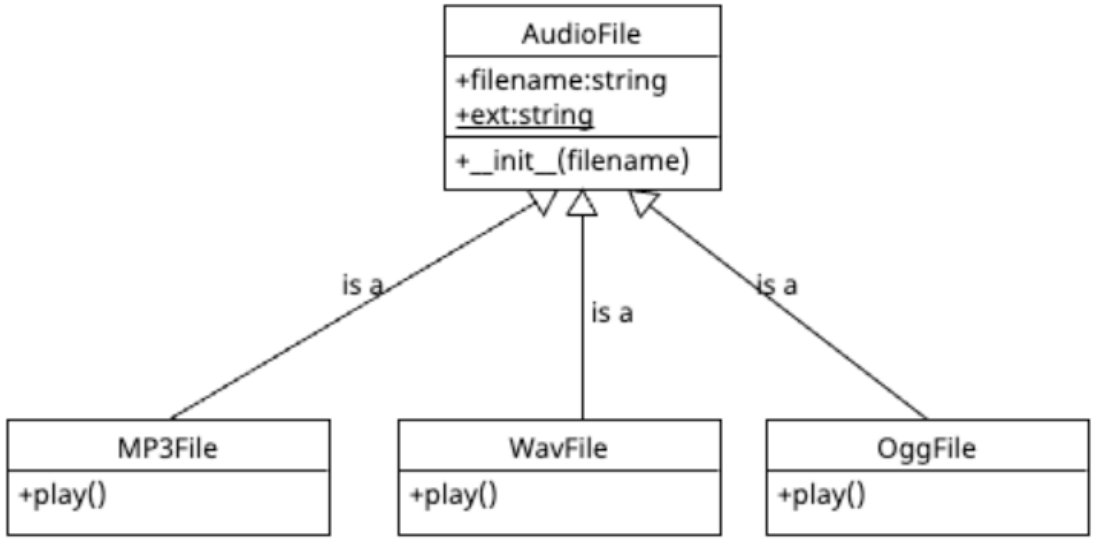
Example:



Different redirections from a superclass AudioFile result in different behaviours when the method `play()` is executed by different subclasses to ensure the correct extraction procedure is followed. The media player object would never need to know which subclass of AudioFile it is referring. It just calls play() method and polymorphically lets the object take care of the actual details of playing.

Polymorphism

```
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):
        print("playing {} as mp3".format(self.filename))


class WavFile(AudioFile):
    ext = "wav"

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


class OggFile(AudioFile):
    ext = "ogg"

    def play(self):
        print("playing {} as ogg".format(self.filename))
```




```
>>> ogg = OggFile("myfile.ogg")
>>> ogg.play()
    playing myfile.ogg as ogg
>>> mp3 = MP3File("myfile.mp3")
>>> mp3.play()
    playing myfile.mp3 as mp3
>>> not_an_mp3=MP3File("myfile.ogg")
    Traceback (most recent call last):
      File "<pyshell#54>", line 1, in <module>
        not_an_mp3=MP3File("myfile.ogg")
      File "<pyshell#43>", line 4, in __init__
        raise Exception("Invalid file format")
    Exception: Invalid file format
```

Polymorphism vs Duck typing

- Polymorphism is one of the coolest things about object-oriented programming, and it makes some programming designs obvious that weren't possible in earlier paradigms.
- However, Python makes polymorphism seem less awesome because of duck typing.
- The name of Duck typing comes from the phrase, “If it walks like a duck and it quacks like a duck, then it must be a duck. So we don’t have to declare it is duck.”
- Duck typing is related to dynamic typing, where the type of the class of an object is going to be less important than the methods that it defines.
- Duck typing in Python allows us to use any object that provides the required behavior without forcing it to be a subclass. The dynamic nature of Python makes this trivial.
- For example, the FlacFile class is not a subclass of AudioFile class, but because it implements the **play()** method, it can be interacted with in Python using the same interface

Polymorphism vs Duck typing

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")

        self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

```
>>> flac = FlacFile("myfile.flac")
>>> flac.play()
    playing myfile.flac as flac
```

- Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts.
- Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses.
- Inheritance can still be useful for sharing code, but if all that is being shared is the public interface, duck typing is all that is required. This reduced need for inheritance also reduces the need for multiple inheritance.
- When multiple inheritance appears to be a valid solution, we can just use duck typing to mimic one of the multiple superclasses.
- Of course, just because an object satisfies a particular interface (by providing required methods or attributes) does not mean it will simply work in all situations. It has to fulfill that interface in a way that makes sense in the overall system.
- Just because an object provides a play() method does not mean it will automatically work with a media player.

Abstract base classes

- While duck typing is useful, it is not always easy to tell in advance if a class is going to fulfil the protocol you require.
- Therefore, Python introduced the idea of abstract base classes (ABCs).
- 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.
- Java also has similar mechanisms, i.e., abstract class and interface.

Abstract base class

- Most of the abstract base classes that exist in the Python standard library live in the ‘collection’ module.
- The ‘Container’ class is one of the simplest ABC

```
>>> from collections.abc import Container
>>> Container.__abstractmethods__
frozenset({'__contains__'})
```

- So the Container class has only one abstract method that needs to be implemented, __contains__.

```
>>> help(Container.__contains__)
Help on function __contains__ in module collections.abc:

__contains__(self, x)
```

Using abstract base class

- We can define a silly containers that tells us whether a given value is in the set of odd integers.
- We can instantiate an 'OddContainer' object without extending Container class.
- However, because it implements the __contains__ method, it is still considered as a subclass of Container in Python.
- That is why duck typing is more popular than classical polymorphism. We can create is a relationships without the overhead of writing the code to set up inheritance.

```
>>> class OddContainer:
...     def __contains__(self,x):
...         if not isinstance(x, int) or not x % 2:
...             return False
...         return True
...
...
>>> odd_container = OddContainer()
>>> isinstance(odd_container, Container)
True
>>> issubclass(OddContainer, Container)
True
```

A more formal way is

```
from abc import ABC

class MyABC(ABC):
    pass
```

Creating your own abstract base class

- In most situations, it is not necessary to have an abstract base class to enable duck typing.
- However, for the documentation purpose, it is advisable to create your own abstract base class.
- The metaclass ABCMeta is used for defining ABCs.
- The term **metaprogramming** refers to the potential for a program to have knowledge of or manipulate itself.
- Python supports a form of metaprogramming for classes called metaclasses.

Creating your own abstract base class

Python decorators:

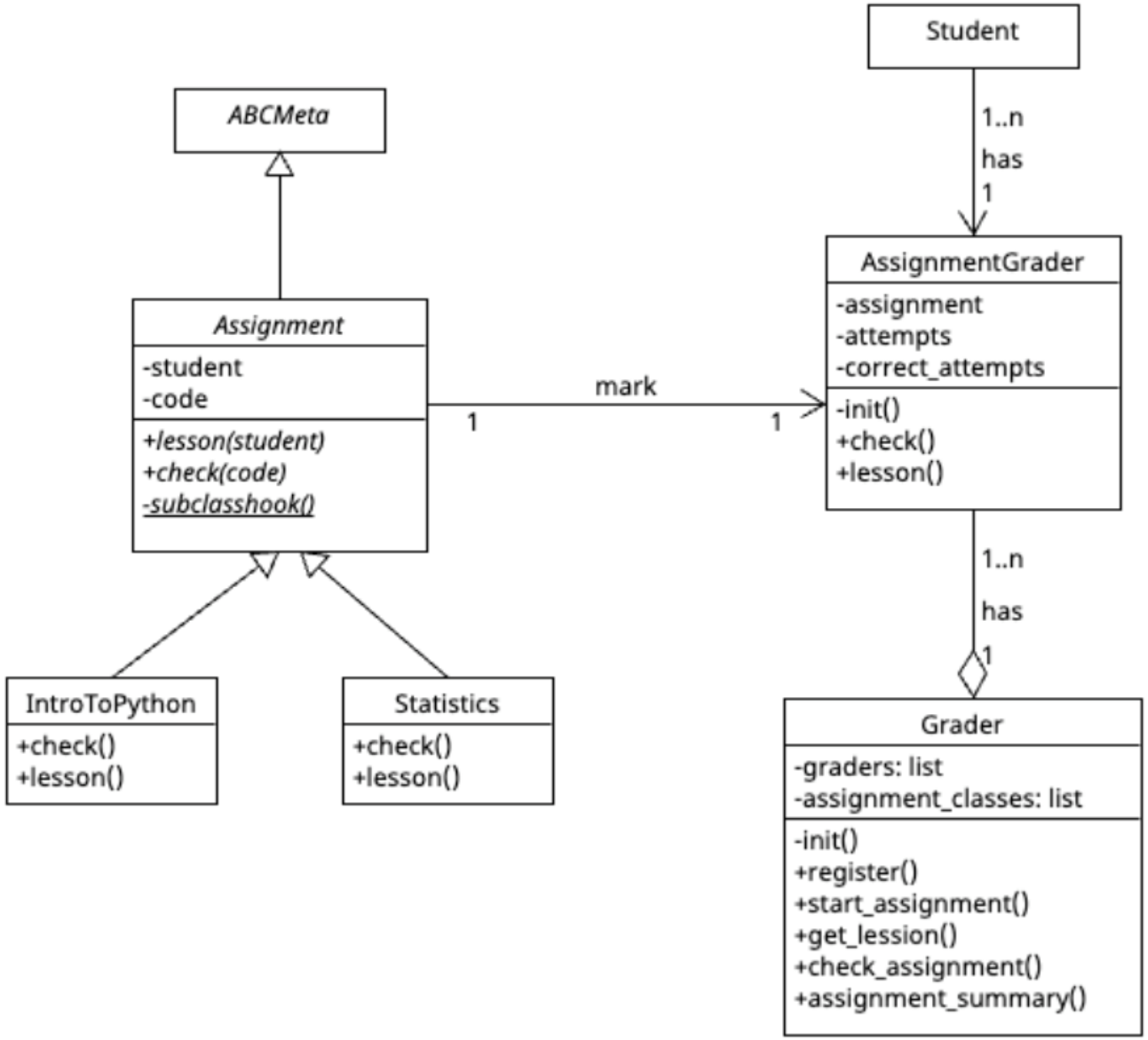
- @abc.**abstractmethod:** indicating abstract methods.
- @**property**: allows a function to be accessed like an attribute
- **@classmethod**: marks the method as a class method.
- **__subclasshook__**(*subclass*): Check whether *subclass* is considered a subclass of this ABC. This method is called by Python interpreter automatically and should return True, False or NotImplemented.
- **attrs = set(dir(C))**: get the set of methods and properties that Class C has, including any parent classes in its class hierarchy.
- **if set(cls.__abstractmethods__) <= attrs**: to see whether the set of abstract methods in this class has been supplied in the candidate class.

```
>>> import abc
>>> class MediaLoader(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def play(self):
...         pass
...
...     @property
...     @abc.abstractmethod
...     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
...
...
>>> class Ogg():
...     ext='.ogg'
...     def play(self):
...         print("this will play an ogg file")
...
...
>>> issubclass(Ogg, MediaLoader)
True
>>> isinstance(Ogg(),MediaLoader)
True
```

A case study

- We will develop an automated grading system for programming assignments.
- The system needs to provide a simple class-based interface for course writers to create their assignments and should give a useful error message.
- The writers need to be able to supply their lesson content and to write custom answer-checking code to make sure students got the right answers.
- The grader will need to keep track of which assignment the student is working on.
- A student might make several attempts at an assignment. We want to keep track of the number of attempts.

A case study: UML class diagram



A case study: implementation

```python
class Assignment(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def lesson(self, student):
        pass

    @abc.abstractmethod
    def check(self, code):
        pass

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

        return NotImplemented
```

```python
class IntroToPython:
    def lesson(self):
        return f"""
            Hello {self.student}. define two variables,
            an integer named a with value 1
            and a string named b with value 'hello'

        """

    def check(self, code):
        return code == "a = 1\nb = 'hello'"


class Statistics(Assignment):
    def lesson(self):
        return (
            "Good work so far, "
            + self.student
            + ". Now calculate the average of the numbers "
            + " 1, 5, 18, -3 and assign to a variable named 'avg'"
        )

    def check(self, code):
        import statistics

        code = "import statistics\n" + code

        local_vars = {}
        global_vars = {}
        exec(code, global_vars, local_vars)

        return local_vars.get("avg") == statistics.mean([1, 5, 18, -3])
```

A case study: implementation

```
class AssignmentGrader:
    def __init__(self, student, AssignmentClass):
        self.assignment = AssignmentClass()
        self.assignment.student = student
        self.attempts = 0
        self.correct_attempts = 0

    def check(self, code):
        self.attempts += 1
        result = self.assignment.check(code)
        if result:
            self.correct_attempts += 1

        return result

    def lesson(self):
        return self.assignment.lesson()
```

```
class Grader:
    def __init__(self):
        self.student_graders = {}
        self.assignment_classes = {}

    def register(self, assignment_class):
        if not issubclass(assignment_class, Assignment):
            raise RuntimeError(
                "Your class does not have the right methods"
            )

        id = uuid.uuid4()
        self.assignment_classes[id] = assignment_class
        return id

    def start_assignment(self, student, id):
        self.student_graders[student] = AssignmentGrader(
            student, self.assignment_classes[id]
        )

    def get_lesson(self, student):
        assignment = self.student_graders[student]
        return assignment.lesson()

    def check_assignment(self, student, code):
        assignment = self.student_graders[student]
        return assignment.check(code)

    def assignment_summary(self, student):
        grader = self.student_graders[student]
        return f"""
        {student}'s attempts at {grader.assignment.__class__.__name__}:

        attempts: {grader.attempts}
        correct: {grader.correct_attempts}

        passed: {grader.correct_attempts > 0}
        """
```

A case study: implementation

Testing

```
from grader import Grader
from lessons import IntroToPython, Statistics

grader = Grader()
itp_id = grader.register(IntroToPython)
stat_id = grader.register(Statistics)

grader.start_assignment("Tammy", itp_id)
print("Tammy's Lesson:", grader.get_lesson("Tammy"))
print(
    "Tammy's check:",
    grader.check_assignment("Tammy", "a = 1 ; b = 'hello'
)
print(
    "Tammy's other check:",
    grader.check_assignment("Tammy", "a = 1\nb = 'hello'"
)

print(grader.assignment_summary("Tammy"))

grader.start_assignment("Tammy", stat_id)
print("Tammy's Lesson:", grader.get_lesson("Tammy"))
print("Tammy's check:", grader.check_assignment("Tammy",
print(
    "Tammy's other check:",
    grader.check_assignment(
        "Tammy", "avg = statistics.mean([1, 5, 18, -3])"
    ),
)

print(grader.assignment_summary("Tammy"))
```

Results

```
= RESTART: /Users/fren/Library/CloudStorage/OneDrive-Universit
yofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/labs/Lab5/
Case Study_ Online Grader/example_runner.py
IntroToPython is Assignment subclass: True
Statistics is Assignment subclass: True
Tammy's Lesson:
            Hello Tammy. define two variables,
            an integer named a with value 1
            and a string named b with value 'hello'


Tammy's check: False
Tammy's other check: True

        Tammy's attempts at IntroToPython:

        attempts: 2
        correct: 1

        passed: True

Tammy's Lesson: Good work so far, Tammy. Now calculate the ave
rage of the numbers  1, 5, 18, -3 and assign to a variable nam
ed 'avg'
Tammy's check: True
Tammy's other check: True

        Tammy's attempts at Statistics:

        attempts: 2
        correct: 2

        passed: True
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 3: When objects are alike

Python

- https://www.python.org/
- https://docs.python.org/3/tutorial/classes.html#inheritance
- https://docs.python.org/3/library/abc.html