Object Oriented Programming in Python
Lesson 2
Inheritance

Outline:

- Basic inheritance
- Inheriting from built-ins
- Multiple inheritance
- Polymorphism
- Abstract base classes

Basic inheritance

Technically, every class we create uses inheritance. All Python classes are subclasses of the special class named "object". A superclass, or parent class, is a class that is being inherited from. A subclass is a class that is inheriting from (derived from or extends) a superclass. That is, the subclass inherits members and methods of the superclass, and we can extend the behaviour of the superclass with new members and new methods in the subclass.

```
class Contact:
    # class variable shared by all instances of the class
    all_contacts = []
    # initializer method
    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

class Supplier(Contact):
    # inherits members and methods of Contact class and has an additional new method
    def order(self, order):
        print("If this were a real system we would send "
        "'{}' order to '{}'".format(order, self.name))
```

Accessing the class variable: Contact.all_contacts

Note: we can also access the class variable as *self.all_contacts* on any object instantiated from Contact. BUT if you ever set the variable using *self.all_contacts*, you will actually be creating a new instance variable associated only with that object.

<u>Testing:</u>

```
c = Contact("Some Body", "somebody@example.net")
print(c.name, c.email)
s = Supplier("Sup Plier", "supplier@example.net")
print(s.name, s.email)
# Both c and s are in the all_contacts list
print(Contact.all_contacts)
# But we can order only from suppliers
c.order("I order sg")  # it does not work
s.order("I order sg")  # it works
```

Extending built-in classes

We can add new functionality to built-in types using inheritance. For example, the ContactList class extends the list type with a search function.

```
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

class Contact:
    # Now we create a ContactList instead of a normal list
    all_contacts = ContactList()
    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

Testing:

```
c1 = Contact("John A", "johna@example.net")
c2 = Contact("John B", "johnb@example.net")
c3 = Contact("Jenna C", "jennac@example.net")
[c.name for c in Contact.all_contacts.search('John')]
```

Most built-in types can be extended. Commonly extended built-ins are object, list, set, dict, file, str, and numerical types such as int and float.

Overriding and Super

Inheritance is also good for changing the behaviour of the superclass. Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. In Python, no special syntax is needed to do this and any method can be overriden (even the __init__ method).

```
class Friend(Contact):
    # We override the __init__ method of the Contact class
    # by adding a new variable to the parameter list
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

Problems:
1. The Friend class has duplicate code to set up the name and email properties (which are also inherited from the Contact class). This can make code maintenance complicated.
2. The Friend class is neglecting to add itself to the all_contacts list we have created on the Contact class.

Solution: Execute the original __init__ method on the Contact class and then add the new member.

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

The super function returns an instance of the parent class, on which we can call the parent method directly. A super() call can be made inside any method, and the call to super can be made at any point in the method; we don't have to make the call as the first line in the method.

Multiple inheritance

In principle, it's very simple: a subclass that inherits from more than one parent class is able to access functionality from both of them. In practice, this is less useful than it sounds and many expert programmers recommend against using it.

The simplest and most useful form of multiple inheritance is called a **mixin**. A mixin is generally a superclass that is not meant to exist on its own, but is meant to be inherited by some other class to provide extra functionality.

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

class EmailableContact(Contact, MailSender):
	# no extra functionality
	pass
```

Testing:

```
e = EmailableContact("John Smith", "jsmith@example.net")
print(Contact.all_contacts)
e.send_mail("Hello, test e-mail here")
```

Multiple inheritance works all right when mixing methods from different classes, but it gets very messy when we have to 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?

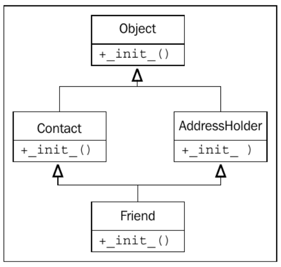
For example, we want to add an address (with multiple strings) to the Friend class. We can solve it using (multiple) inheritance, or composition.

Solution using inheritance

```
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
```

The diamond problem: a superclass may be called multiple times because of the organization of the class hierarchy.



Solution using composition

```
class Address:
        def __init__(self, number, street, city, code):
                self.number = number
                self.street = street
                self.city = city
                self.code = code

class Friend(Contact):
        def __init__(self, name, email, phone, number, street, city, code):
                Contact.__init__(self, name, email)
                self.phone = phone
                self.adr = Address(number, street, city, code)
```

Composition is the technique for implementing has_a relationship between objects. Here, each Friend object will have an Address object as member. In this way we can avoid multiple inheritance. (Composition over Inheritance).

Polymorphism

It means that different behaviors happen depending on which subclass is being used. We can use inheritance with polymorphism to simplify the design.

Example: add a send_email function to Contact and override it in Supplier and Friend in different ways. The version of the send_email function to be called depends on the type of instance that activates the function.

Duck typing

In nominative typing, an object is *of a given type* if it is declared to be through inheritance. In duck typing, an object is *of a given type* if it has all methods and properties required by that type.

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. In practice, it's rarely necessary to create new abstract base classes, since we can use existing ones. Furthermore, it's not necessary to have an abstract base class to enable duck typing.

<u>Using existing ABC</u>

Most of the abstract base classes that exist in the Python Standard Library live in the collections module. One of the simplest ones is the Container class. The Container class has exactly one abstract method that needs to be implemented, __*contains*__.

Let's define a container that tells us whether a given value is in the set of odd integers:

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

The OddContainer class does not extend the Container class, but implements all its methods. Therefore the OddContainer class can be considered as a duck type instance of the Container class.

```
from collections import Container
odd_container = OddContainer()
isinstance(odd_container, Container)
>>>True
issubclass(OddContainer, Container)>
>>>True
```

Note: Any class that has a __contains__ method is a Container and can therefore be queried by the 'in' keyword.

```
1 in odd_container
>>>True
2 in odd_container
>>>False
```