Inheritance

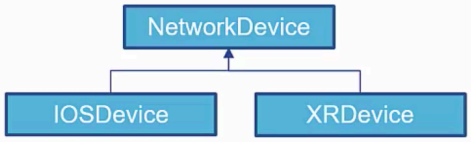
- **Parent** or **base class**

Contains basic attributes common to all objects that *inherit* from that base class.

- **Child** or **sub class**

Contains values that are specific to that type of object

- Relationship: "**is-a**"




```
class NetworkDevice():
   # general data: ip,user,pw
   # general methods

class IOSDevice(NetworkDevice):
   # IOS-specific data
   # IOS-specific methods

class XRDevice(NetworkDevice):
   # XR-specific data
   # XR-specific methods

...
```

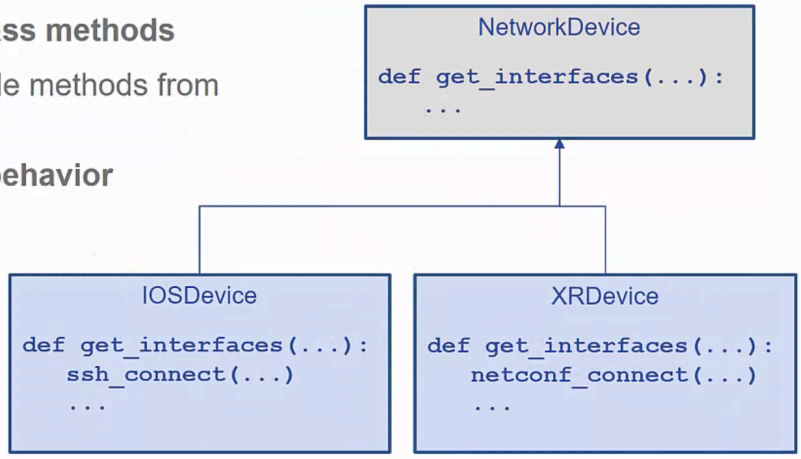

Overriding Methods

- **Overriding base-class methods**

Subclasses can override methods from base class if required

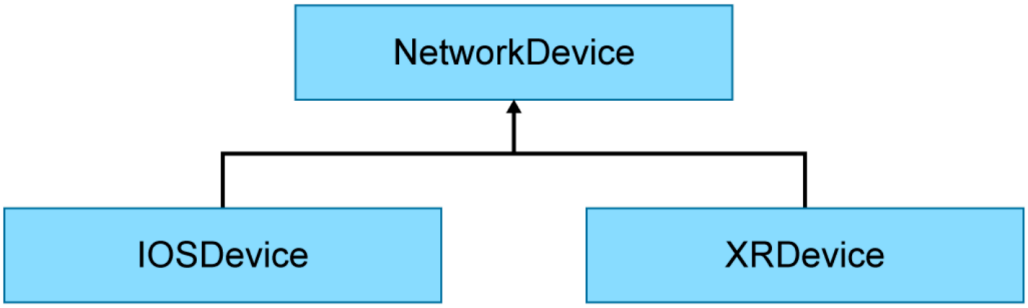
- **Subclass-specific behavior**

Allows subclasses to implement specific behavior unique to that subclass



In inheritance, the basic idea is that there is a 'base' or 'parent' class, which represents the general data and functionality for the most general definition of that type of object. This parent class provides the basis from which child classes are derived. The derived classes inherit the attributes and functions of the base class, and they add their own unique attributes and functions which are particular to that child type of object.

The relationship between parent and child class is referred to as an "is-a" relationship, as in "a circle is a shape." In a networking context, "a router is a network device." Other networking examples include "a gigabit Ethernet interface is an interface," and "a link-state topology is a topology."



Consider the following snippets of code:

```
class NetworkDevice():
   # general data: ip,user,pw
   # general methods
class IOSDevice(NetworkDevice):
   # IOS-specific data
   # IOS-specific methods
class XRDevice(NetworkDevice):
   # XR-specific data
   # XR-specific methods
```

In the code above, notice:

- **Base class:** The class `NetworkDevice` is the base class for all types of networking devices. One can imagine child classes for various generic device types, such as router, switch, wireless device, firewall, WAN optimizer, and so on.

  The example is creating just one level of inheritance, and defining child classes which are either IOS devices or IOS-XR devices.

  The base class will contain attributes that are common to all children of its type. For example, all networking devices will have attributes for the device name, device management IP address, device administrator username, password, and so on. And there will likely be methods common to all networking devices, such as to provide the ability to set or retrieve these values.

- **Child classes:** The example defined two child classes, one for IOS devices, one for IOS-XR devices. As shown in the comments in the code, there will be attributes (data) and methods that are specific to each device type.

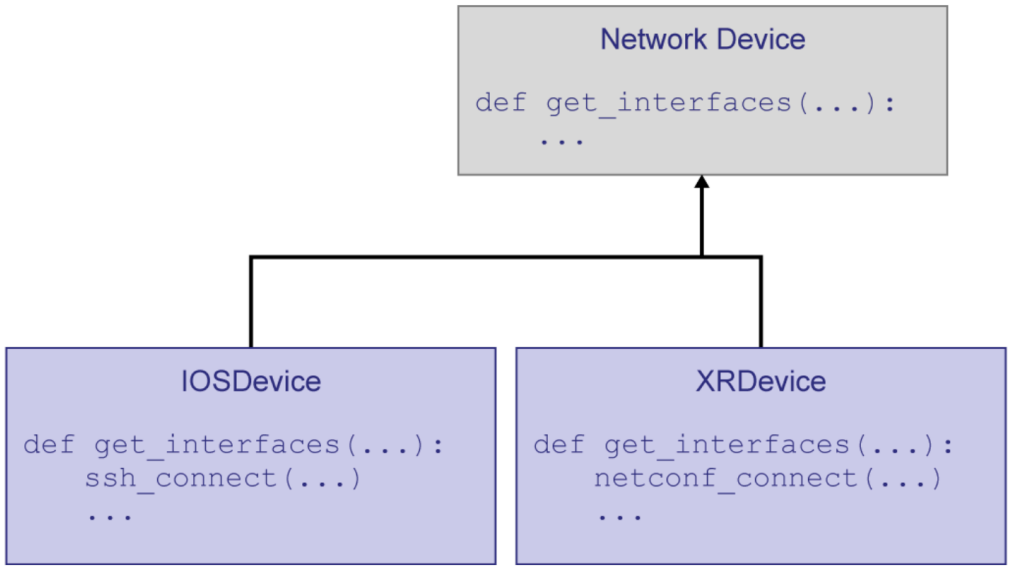
  Child classes are derived from a base class, and the definition of the child class references its parent class in parentheses at the top of the class definition. In the example:

  ```
  class IOSDevice(NetworkDevice):
  The class IOSDevice above derives from the class NetworkDevice.
  ```

  The class definition for each type of device may contain device-specific information (for example, IOS-XR routing capabilities, IOS switching capabilities). For the IOS-XR type of device, there may be routing attributes and the ability to call methods to set static routes. For the IOS type of device, there may be switching attributes and the ability to set parameters for spanning tree and edge security.

Overriding Methods

The next important aspect of inheritance is the ability to override methods in each child class. The general idea is that a common method may be defined in the base class, but each child class will define device-specific implementations of that method.



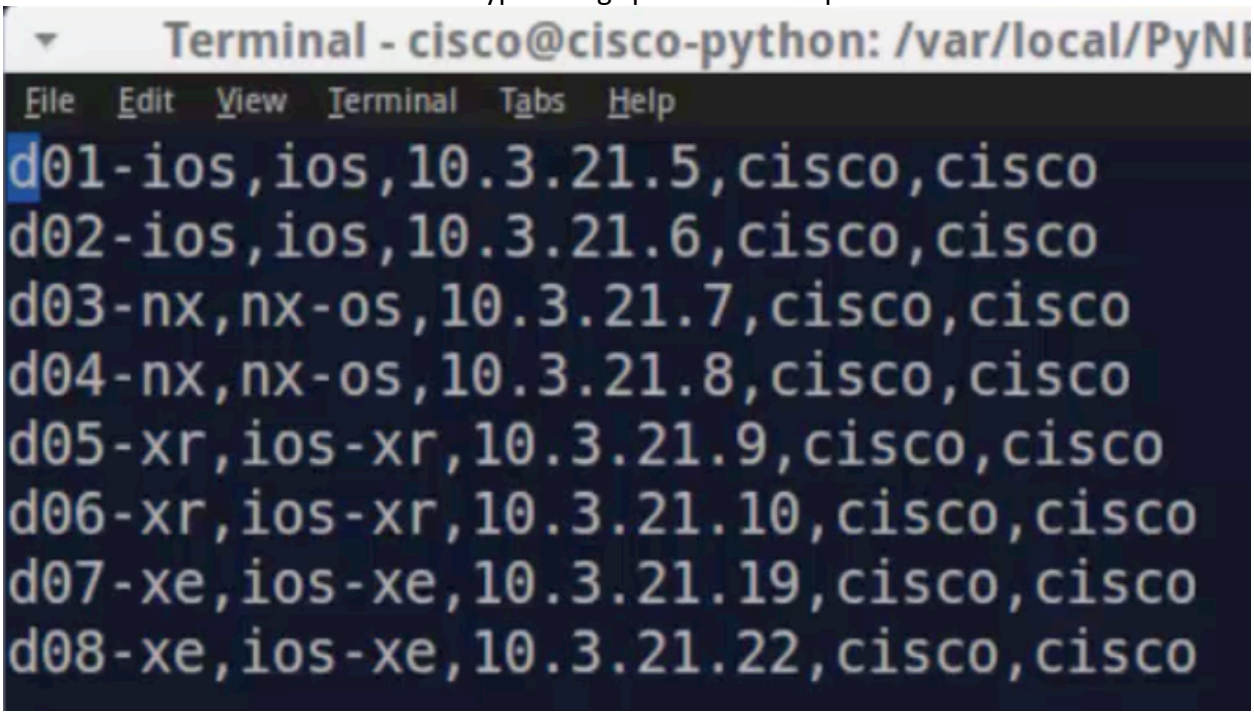
Consider the following code:

```
class NetworkDevice():
    def get_interfaces(self,...):
        return None

class IOSDevice(NetworkDevice):
    def get_interfaces(self,...):
        session = ssh_connect(...)
        interfaces = show_int_brief(self.ip,...)
        return interfaces

class XRDevice(NetworkDevice):
    def get_interfaces(self,...):
        session = netconf_connect(...)
        interfaces = get_netconf_oper_int(self.ip,...)
        return interfaces
```

In the code above, the base NetworkDevice class defines a method called get_interfaces(...). But each sub-class implements its own version of that method. In the simplistic example above, the IOS device implements get_interfaces using SSH, whereas the XR device, which supports NETCONF, is able to satisfy the request using that protocol.

Overriding Methods

- We override the base the os type string specified in the parent



```
#---- Class to hold information about a generic network device --------
class NetworkDevice():

    def __init__(self, name, ip, user='cisco', pw='cisco'):
        self.name = name
        self.ip_address = ip
        self.username = user
        self.password = pw

    def get_type(self):
        return 'base'

#---- Class to hold information about an IOS-XE network device --------
class NetworkDeviceIOS(NetworkDevice):

    def __init__(self, name, ip, user='cisco', pw='cisco'):
        NetworkDevice.__init__(self, name, ip, user, pw)

    def get_type(self):
        return 'IOS'
#---- Class to hold information about an IOS-XR network device --------
class NetworkDeviceXR(NetworkDevice):

    def __init__(self, name, ip, user='cisco', pw='cisco'):
        NetworkDevice.__init__(self, name, ip, user, pw)

    def get_type(self):
        return 'IOS-XR'
```

```
def read_device_info(devices_file):

    devices_list = []

    # Read in the devices from the file
    file = open(devices_file,'r')
    for line in file:

        device_info = line.strip().split(',') # Get device info into list

        # Create a device object with this data
        if device_info[1] == 'ios':

            device = NetworkDeviceIOS(device_info[0],device_info[2],
                                      device_info[3],device_info[4])

        elif device_info[1] == 'ios-xr':

            device = NetworkDeviceXR(device_info[0],device_info[2],
                                     device_info[3],device_info[4])

        else:
            device = NetworkDevice(device_info[0],device_info[2],
                                   device_info[3],device_info[4])

        devices_list.append(device) # add this device object to list

    file.close() # Close the file since we are done with it

    return devices_list
#---- Function to go through devices printing them to table -----------
def print_device_info(devices_list):

    print ''
    print 'Name     OS-type  IP address       Username  Password'
    print '------   -------  --------------   --------  --------'

    # Go through the list of devices, printing out values in nice format
    for device in devices_list:

        print '{0:8} {1:8} {2:16} {3:9} {4:9}'.format(device.name,
                                                      device.get_type(),
                                                      device.ip_address,
                                                      device.username,
                                                      device.password)

    print ''
```

```
#---- Main: read device info, then print ------------------------------

devices = read_device_info('devices')
print_device_info(devices)
cisco@cisco-python:/var/local/PyNE/labs/sections/section13$ python S07-4-override.py

Name     OS-type  IP address       Username  Password
------   -------  -------------    --------  --------
d01-ios  IOS      10.3.21.5        cisco     cisco
d02-ios  IOS      10.3.21.6        cisco     cisco
d03-nx   base     10.3.21.7        cisco     cisco
d04-nx   base     10.3.21.8        cisco     cisco
d05-xr   IOS-XR   10.3.21.9        cisco     cisco
d06-xr   IOS-XR   10.3.21.10       cisco     cisco
d07-xe   base     10.3.21.19       cisco     cisco
d08-xe   base     10.3.21.22       cisco     cisco
```