03 Object-Oriented Python

May 26, 2021

Information: *Main concepts and examples of object-oriented python, what may of today's software tools for deep learning are based on*

Written by: *Zihao Xu*

Last Update Date: 05.26.2021

1 Basic Concepts

1.1 Definition of Class

- At a high level of conceptualization, a class can be thought of as a category
- For the purpose of writing code, a class is a data structure with **attributes**
  - In python, the word **attribute** is used to describe any *property,variable* or *method*, that can be invoked with the dot operator on either the class or an instance constructed from a class
- An **instance** constructed from a class will have specific values for the attributes
- To endow instances with behaviors, a class can be provided with **methods**

1.2 Methods

- A method is a function to be invoked on an instance of the class **or** the class itself
- A method that is invoked on an instance is sometimes called an **instance method**
- A method that is invoked directly on a class is called a **class method** or a **static method**
- To define it formally, a **method** is a **function** that can be invoked on an object using the *object-oriented call syntax* that for python is of the form **object.method()**, where *object* may either be an instance of a class or the class itself

1.3 Function Objects vs. Callables

- Python makes an distinction between **function objects** and **callables**. While all function objects are callables, not all callables are function objects
- A **function object** can only be created with a **def** statement
- A **callable** is any object that can be called like a function
  - A class name can be called directly to yield an instance of a class
  - An instance object can also be called directly; what that yields depends on whether or not underlying class provides a definition for the **system-supplied ___call___** method
- In Python, '()' is an Operator - the Function Call Operator
  - E.g. for a call **X** with method **foo**
    - calling just **X.foo** returns the method object itself that **X.foo** stands for

    - calling **X.foo()** cause execution of the function object associated with the method call

1.4 Pre-Defined Attributes

- A class in Python comes with certain **pre-defined attributes**
- The **pre-defined attributes**, both variables and methods, employ a special naming convention: *the names begin and end with two underscores*
- The *pre-defined attributes* of a class are not to be confused with the **programmer-supplied attributes** such as the class and instance variables and the programmer-supplied methods
- May think of the *pre-defined* attributes as the *external properties* of classes and instances and the *programmer-supplied* attributes as the *internal properties*
- Commonly used pre-defined attributes:

1.5 Instance variables, Class variables

- Attributes that take data values on a per-instance basis are frequently referred to as **instance variables**
- Attributes that take data values on a per-class basis are called **class variables**, *static attributes* or *class attributes*

1.6 Encapsulation

- Hiding or controlling access to the implementation-related attributes and the methods of a class is called **encapsulation**
- With appropriate data encapsulation, a class will present a well-defined **public interface** fro its **clients**, the uses of the class
- A client should only access those data attributes and invoke those methods that are in the *public interface*
- As opposed to OO in C++ and Java, all of the attributes defined for a class are **available to all** in Python
    - A Python class and a Python instance object are so open that they can be modified after the objects are brought into existence

1.7 Inheritance

- **Inheritance** in object-oriented code allows a subclass to inherit some or all of the attributes and methods of its superclass(es)

1.8 Polymorphism

- **Polymorphism** basically means that a given category of objects can exhibit multiple identities **at the same time**.
- Allows us to manipulate instances belonging to the different classes of a **hierarchy** through a common interface defined for the **root class**.

2 Defining a Class in Python

2.1 Syntax for defining a class and create an instance

[1]:
```
# ---------- Definition ---------
class Bird:
    alive = True
    is_animal = True


# --------- End of definition ----

# Create an instance in the defined class
bird_1 = Bird()

# Use dot operator to access variables in the instance
print(bird_1.alive, bird_1.is_animal)
```

```
True True
```

2.2 Syntax for methods of a class

- A method defined for a class must have special syntax that *reserves the first parameter for the object on which the method is invoked.* This parameter is typically names **self** for instance methods, but could be any legal Python identifier
  - Usually, any instance variables would be in the syntax **self.variable**
- **Only one method per name in a class** regardless of the parameter structure of the function
  - All method names are stored as keys in the namespace dictionary while the dictionary keys must be unique, so there can exist only one function object for a given method name
  - A more general case is that a class can have **only one attribute of a given name**
- Defining a method outside the class
  - It is not necessary for the body of a method to be enclosed by a class
  - A function object created outside a class can be assigned to a name inside the class.
    - The name will acquire the function object as its binding.
    - Subsequently, that name can be used in a method call as if the method had been defined inside the class

[2]:
```
# ---------- Definition of method ----------
def total(self, arg1, arg2):
    self.N = arg1 + arg2


# ---------- Definition of class -----------
class X:
    # Assign the function object to foo
    foo = total

    # Define a method returns the value N
```

```
    def get_N(self):
        return self.N


# --------- End of Definition --------------

# Create an instance with initial value 10
X_obj = X()
# Call the foo method in the class
X_obj.foo(20, 30)
print(X_obj.get_N())
```

```
50
```

2.3 ___init___() method

- A pre-defined attribute to initialize the instance returned by a call to the constructor
- It will be invoked automatically when creating an instance by calling **class()**
- The parameter *self* in the call to ___init___ is set implicitly to the instance under construction
- If you do not provide a class with its own ___init___, the system will provide the class with a **default** ___init___(). You override the default definition by providing your own implementation for ___init___

[3]:
```
# ---------- Definition ---------
class Person:
    # Descriptions of the class
    "A very simple class"
    # These variables are class variables
    # All instances in this class share the same value for these varibles when␣
 ↪constructed
    alive = True
    is_animal = True

    # Customize the __init__() method
    def __init__(self, name, age):
        # These variables are instance variables
        # The value are to be assigned for every instance to be contructed
        self.name = name
        self.age = age


# --------- End of definition ----

# Create two instances in the defined class
person_1 = Person("John", 20)
person_2 = Person("Alice", 18)

# See the user-supplied attributes of the two instances
```

```
print(person_1.name, person_1.age, person_1.alive, person_1.is_animal)
print(person_2.name, person_2.age, person_2.alive, person_2.is_animal)
```

```
John 20 True True
Alice 18 True True
```

2.4 Defining Static Attributes for a Class

- As class definition usually includes two different kinds of attributes
  - attributes exist on a per-instance basis
  - attributes exist on a per-class basis, which are commonly referred to be as being **static**
- A variable becomes **static** is it is declared outside of any method in a class definition
- For a method to become static, it needs the **staticmethod()** wrapper
  - The modern approach to achieving the same effect is through the use of the **staticmethod decorator**

[4]:
```
# ---------- Definition ---------
class X:
    id_num = 0

    @staticmethod
    def get_id():
        return X.id_num

    @classmethod
    def get_next_id(self):
        X.id_num += 1
        return X.id_num

    def __init__(self, owner):
        self.owner = owner
        self.id = X.get_next_id()


# --------- End of definition ----

# Example of using static method and static variables
# Notice that static method can be directly called without instances
print("Number of instances: %d" % (X.get_id()))
X_1 = X("owner 1")
print("Create an instance owned by %s with ID %d" % (X_1.owner, X_1.id))
X_2 = X("owner 2")
print("Create an instance owned by %s with ID %d" % (X_2.owner, X_2.id))
# Static method can also be invoked on instances
print("Number of instances: %d" % (X_1.get_id()))
```

```
Number of instances: 0
Create an instance owned by owner 1 with ID 1
Create an instance owned by owner 2 with ID 2
```

```
Number of instances: 2
```

2.5 Making a Class Instance Iterable

- A class instance is iterable if you can loop over the data stored in the instance. The data may be stored in the different attributes and in ways not directly accessible by array like indexing
- A class must provide for an iterator in order for its instances to be iterable and this iterable must be returned by the definition of **___iter()___** for the class
- Commonly, the iterator defined for a class will itself be a class that provides implementation for a method named **___next___** in Python3

[5]:
```
import random

random.seed(0)


# Define the class
class X:
    # Initialize with an array
    def __init__(self, arr):
        self.arr = arr

    # Get the desired number with index
    def get_num(self, i):
        return self.arr[i]

    # When called with () operator,return the array
    def __call__(self):
        return self.arr

    def __iter__(self):
        return Xiterator(self)


# Define the iterator
class Xiterator:
    def __init__(self, X_obj):
        self.items = X_obj.arr
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        if self.index < len(self.items):
            return self.items[self.index]
        else:
```

```
            raise StopIteration


# Create an instance
X_obj = X(random.sample(range(1, 10), 5))
print(X_obj.get_num(0))
print(X_obj())
# Iteration over the instance
for item in X_obj:
    print(item, end=" ")
print("")

# Create another instance of the iterator over the same instance of X
iters = iter(X_obj)
for ii in range(len(X_obj())):
    print(iters.__next__(), end=" ")
```

```
7
[7, 9, 1, 3, 5]
7 9 1 3 5
7 9 1 3 5
```

3 How Python Creates and Destructs an Instance

3.1 Create an Instance from a Class

- **Step 1**:
  - The call to the constructor creates what may be referred to as a generic instance from the class definition
  - The generic instance's **memory allocation** is customized with the code in the method **___new___()** of the class. This method may either be defined directly for the class or the class may inherit it from one of its parent classes
  - The method **___new___()** is implicitly considered by Python to be a static method. Its first parameter is meant to be set equal to the name of the class whose instance is desired and it must return the instance created
  - If a class does not provide its own definition for **___new___()**, a search is conducted for this method in the parent classes of the class.
- **Step 2**:
  - Then the instance method **___init___()** of the class is invoked to initialize the instance returned by **___new___()**
- Notice:
  - We do not need any special declaration for **___new___()** to be recognized as static because this method is special-cased by Python

[6]:
```
# ---------- Definition ---------
class X:
    def __new__(cls):
        print("__new__ invoked")
```

```
        return object.__new__(cls)

    def __init__(self):
        print("__init__ invoked")


# --------- End of definition ----

# Create an instance
X_obj = X()
print(dir(X_obj))
```

```
__new__ invoked
__init__ invoked
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
```

3.2 Destruction of Instance Objects

- Python comes with an automatic garbage collector. Each object created is kept track of reference counting. Each time an object is assigned to a variable, its reference count goes up by one, signifying the fact that there is one more variable holding a reference to the object
- And each time a variable whose referent object either goes out of scope or is changed, the reference count associated with the object is decreased by one. When the reference count associated with an object goes to zero, it becomes a candidate for garbage collection
- Python provides us with **___del___** for cleaning up beyond what is done by automatic garbage collection

[7]:
```
# ---------- Definition ---------
class X:
    def __init__(self):
        print("__init__ invoked")

    def __del__(self):
        print("__del__ invoked")


# --------- End of definition ----

# Create an instance
X_obj = X()
print(dir(X_obj))
# Delete X_obj
del X_obj
print(dir(X_obj))
```

```
__init__ invoked
['__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
__del__ invoked


␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-7-c072168ef5c0> in <module>
 13 # Delete X_obj
 14 del X_obj
---> 15 print(dir(X_obj))


NameError: name 'X_obj' is not defined
```

4 Pre-defined Attributes

4.1 Pre-defined Attributes for a Class

- Being an object in its own right, every Python class comes equipped with the following **pre-defined attributes**
  - ___name___: string name of the class
  - ___doc___: documentation string for the class
  - ___bases___: tuple of parent classes of the class
  - ___dict___: dictionary whose keys are the names of the **class variables** and the **methods** of the class and whose values are the corresponding binding
  - ___module___: module in which the class is defined
- As an alternative to invoking ___dict___ on a class name, one can also use the built-in global **dir()** that returns a list of all the attribute names, for variables and for methods, for the class (both directly defined for the class and inherited from a class's superclasses)

[8]:
```
# Print out the class attributes
print(Person.__name__)
print(Person.__doc__)
print(Person.__bases__)
print(Person.__module__)
print(Person.__dict__)
print(dir(Person))
```

```
Person
```

```
A very simple class
(<class 'object'>,)
__main__
{'__module__': '__main__', '__doc__': 'A very simple class', 'alive': True,
'is_animal': True, '__init__': <function Person.__init__ at 0x0000025C57082EE0>,
'__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__':
<attribute '__weakref__' of 'Person' objects>}
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'alive', 'is_animal']
```

4.2 Predefined Attributes for an instance of the class

- Since every class instance is also an object in its own right, it also comes equipped with certain pre-defined attributes. Usually the following two are of particular interests
  - ___class___: string name of the class form which the instance was constructed
  - ___dict___: dictionary whose keys are the names of the **instance variable**
    - Notice: It's important to realize that the namespace as represented by the dictionary ___dict___ for a **class object** is not the same as the namespace as represented by the dictionary ___dict___ for an **instance object** constructed from the class
- When called on an instance, **dir()** will return the same list as above plus any instance variables defined for the class

[9]:
```
# Print out instance attributes
print(person_1.__class__)
print(person_1.__dict__)
print(dir(person_1))
```

```
<class '__main__.Person'>
{'name': 'John', 'age': 20}
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'age', 'alive', 'is_animal',
'name']
```

4.3 The Root Class *object*

- All classes are subclassed, either directly or indirectly from the root class **object**
- The **object** class defines a set of methods with default implementations that are inherited by all classes derived from object
- The list of attributes defined for the **object** class can be seen by printing out the list returned by the built-in **dir()** function
- Alternatively, the ___dict___ attribute will provide both the attribute names and their binding

[10]:
```
# Print out the attributes of root class - object
print(dir(object))
print(object.__dict__)
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
{'__repr__': <slot wrapper '__repr__' of 'object' objects>, '__hash__': <slot
wrapper '__hash__' of 'object' objects>, '__str__': <slot wrapper '__str__' of
'object' objects>, '__getattribute__': <slot wrapper '__getattribute__' of
'object' objects>, '__setattr__': <slot wrapper '__setattr__' of 'object'
objects>, '__delattr__': <slot wrapper '__delattr__' of 'object' objects>,
'__lt__': <slot wrapper '__lt__' of 'object' objects>, '__le__': <slot wrapper
'__le__' of 'object' objects>, '__eq__': <slot wrapper '__eq__' of 'object'
objects>, '__ne__': <slot wrapper '__ne__' of 'object' objects>, '__gt__': <slot
wrapper '__gt__' of 'object' objects>, '__ge__': <slot wrapper '__ge__' of
'object' objects>, '__init__': <slot wrapper '__init__' of 'object' objects>,
'__new__': <built-in method __new__ of type object at 0x00007FFF9D6FBB50>,
'__reduce_ex__': <method '__reduce_ex__' of 'object' objects>, '__reduce__':
<method '__reduce__' of 'object' objects>, '__subclasshook__': <method
'__subclasshook__' of 'object' objects>, '__init_subclass__': <method
'__init_subclass__' of 'object' objects>, '__format__': <method '__format__' of
'object' objects>, '__sizeof__': <method '__sizeof__' of 'object' objects>,
'__dir__': <method '__dir__' of 'object' objects>, '__class__': <attribute
'__class__' of 'object' objects>, '__doc__': 'The base class of the class
hierarchy.\n\nWhen called, it accepts no arguments and returns a new
featureless\ninstance that has no instance attributes and cannot be given
any.\n'}
```

5 Multiple-Inheritance Class Hierarchies

5.1 Extending a Class

- The syntax of **calling a Superclass Method Definition Directly** in the following codes, in the example:
  - Both the **promote()** and **myprint()** methods of the derived class to call on the method of the same names in the base class to do part of the job
  - It is accomplished by calling the methods promote() and the myprint() directly on the class name Employee

[11]:
```
# base class Employee
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
```

```
    promotion_table = {
        "shop_floor": "staff",
        "staff": "manager",
        "manager": "executive"
    }

    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print(self.name, "%s" % self.position, end=" ")


# Derived class Manager
class Manager(Employee):
    def __init__(self, nam, pos, dept):
        # Call the __init__ method in Employee class directly
        Employee.__init__(self, nam, pos)
        self.dept = dept

    def promote(self):
        if self.position == "executive":
            print("not possible")
            return
        # Call the promote method in Employee class directly
        Employee.promote(self)

    def myprint(self):
        Employee.myprint(self)
        # Call the myprint method in Employee class directly
        print(self.dept)


# Create an employee "Orpheus" with initial position "staff" from the base class
emp = Employee("Orpheus", "staff")
emp.myprint()
print("")
# Use the method in base class to promote this employee
emp.promote()
print(emp.position)
emp.myprint()
print("\n")

# Create a manager "Zaphod" with initial position "manager" in department␣
 ↪"sales" from the derived class
man = Manager("Zaphod", "manager", "sales")
man.myprint()
```

```
man.promote()
print(man.position)
man.myprint()
print("")

# See the relationship between base/derived instances and base/derived classes
# The derived instance would be an instance in the base class
print(isinstance(man, Employee))
# The base instance would not be an instance in the derived class
print(isinstance(emp, Manager))
# All instances are in the object class
print(isinstance(man, object))
print(isinstance(emp, object))
```

```
Orpheus staff
manager
Orpheus manager

Zaphod manager sales
executive
Zaphod executive sales

True
False
True
True
```

- The methods of the derived class can invoke the built-in **super()** for calling on the methods of the parent class

[12]:
```
# base class Employee
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos

    promotion_table = {
        "shop_floor": "staff",
        "staff": "manager",
        "manager": "executive"
    }

    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print(self.name, "%s" % self.position, end=" ")
```

```python
# Derived class Manager
class Manager(Employee):
    def __init__(self, nam, pos, dept):
        # Use super() to call the __init__ method in Employee class
        super().__init__(nam, pos)
        self.dept = dept

    def promote(self):
        if self.position == "executive":
            print("not possible")
            return
        # Use super() to call the promote method in Employee class
        super(Manager, self).promote()

    def myprint(self):
        # Use super() to call the myprint method in Employee class
        super(Manager, self).myprint()
        print(self.dept)


# Create an employee "Orpheus" with initial position "staff" from the base class
emp = Employee("Orpheus", "staff")
emp.myprint()
print("")
# Use the method in base class to promote this employee
emp.promote()
print(emp.position)
emp.myprint()
print("\n")

# Create a manager "Zaphod" with initial position "manager" in department␣
 ↪"sales" from the derived class
man = Manager("Zaphod", "manager", "sales")
man.myprint()
man.promote()
print(man.position)
man.myprint()
```

```
Orpheus staff
manager
Orpheus manager

Zaphod manager sales
executive
Zaphod executive sales
```

5.2 Derive a class from multiple base classes

- Python allows a class to be derived from multiple base classes
- The order in which the class and its bases are searched for the implementation code is commonly referred to as the **Method Resolution Order (MRO)** in Python
- The following example shows the syntax for multiple inheritance

[13]:
```
class A(object):
    def __init__(self):
        print("called A's init")


class B(A):
    def __init__(self):
        print("called B's init")
        super().__init__()


class C(A):
    def __init__(self):
        print("called C's init")
        super().__init__()


class D(B, C):
    def __init__(self):
        print("called D's init")
        super().__init__()


D_obj = D()
print(D.__mro__)
```

```
called D's init
called B's init
called C's init
called A's init
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>)
```

An example applying basic OO concepts 1. Create a class named **People** that has three instance variables named: - first_names - middle_names - last_names 2. Expand the class definition and endow it with an iterator, iterating through the data stored in the **People** instance should print out the 10 names in "first_name second_name third_name" order 3. Further expand the definition of the class and endow it with another variable that takes one of the following three values. When iterating through the instance, it prints out the names in the chosen format. - first middle last - last first middle - last, first middle 4. Further expand the definition of the class and make its instances callable. When the function-call operator "()" is applied, it shoud print out a sorted list of just the last names

[14]:
```
# ----------------------------- class People -----------------------------
class People:
    def __init__(self, nam1, nam2, nam3, order):
        self.first_names = nam1
        self.middle_names = nam2
        self.last_names = nam3
        self.order = order

    def __iter__(self):
        return PeopleIterator(self)

    def __call__(self):
        index = -1
        for _ in range(len(self.last_names)):
            index += 1
            print(self.last_names[index], end='\n')
        return self.last_names


class PeopleIterator:
    def __init__(self, PeopleObject):
        self.List1 = PeopleObject.first_names
        self.List2 = PeopleObject.middle_names
        self.List3 = PeopleObject.last_names
        self.order = PeopleObject.order
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        NameList = []
        NameList2 = []
        if self.index < len(self.List1):
            if self.order == 'first_name_first':
                NameList.append(self.List1[self.index])
                NameList.append(self.List2[self.index])
                NameList.append(self.List3[self.index])
                Name = " ".join(NameList)
            elif self.order == 'last_name_first':
                NameList.append(self.List3[self.index])
                NameList.append(self.List1[self.index])
                NameList.append(self.List2[self.index])
                Name = " ".join(NameList)
            elif self.order == 'last_name_with_comma_first':
                NameList.append(self.List3[self.index])
```

```
                NameList.append(self.List1[self.index])
                NameList2.append(", ".join(NameList))
                NameList2.append(self.List2[self.index])
                Name = " ".join(NameList2)
            else:
                print("Name order is not supported type")
                Name = None
        else:
            raise StopIteration
        return Name


# ------------------------ end of class definition ------------------------
```

[15]:
```
import random
import string

#  Generate an array of 10 random strings of length 5 for each variables
random.seed(0)
FirstList = []
MiddleList = []
LastList = []
for ii in range(10):
    Randomlist1 = []
    Randomlist2 = []
    Randomlist3 = []
    for jj in range(5):
        Randomlist1.append(random.choice(string.ascii_lowercase))
        Randomlist2.append(random.choice(string.ascii_lowercase))
        Randomlist3.append(random.choice(string.ascii_lowercase))
    Randomstring1 = "".join(Randomlist1)
    Randomstring2 = "".join(Randomlist2)
    Randomstring3 = "".join(Randomlist3)
    FirstList.append(Randomstring1)
    MiddleList.append(Randomstring2)
    LastList.append(Randomstring3)

# Create three instances of the People class storing the random strings in␣
 ↪different orders
people1 = People(FirstList, MiddleList, LastList, 'first_name_first')
people2 = People(FirstList, MiddleList, LastList, 'last_name_first')
people3 = People(FirstList, MiddleList, LastList, 'last_name_with_comma_first')

# Iterating through the data stored in People instance in different orders
print("In the order of %s:" % (people1.order))
for item in people1:
    print(item, end="\n")
```

```
print("")

print("In the order of %s:" % (people2.order))
for item in people2:
    print(item, end="\n")
print("")

print("In the order of %s:" % (people3.order))
for item in people3:
    print(item, end="\n")
print("")

# Print out the sorted last names by applying the function-call operator
people1()
print("")
```

```
In the order of first_name_first:
mbpjs yimpg nqzlq
eyzwe jdizj etrtd
xkdkg cpltr vrnup
obamv qzcwu irxza
thwsz pxche kkghz
rcprd okdjr cqjwk
rtjtk grozs zscmh
jfvcy fbice gtpve
brwqg cvqzv wmihs
noult spwcd ivzkp

In the order of last_name_first:
nqzlq mbpjs yimpg
etrtd eyzwe jdizj
vrnup xkdkg cpltr
irxza obamv qzcwu
kkghz thwsz pxche
cqjwk rcprd okdjr
zscmh rtjtk grozs
gtpve jfvcy fbice
wmihs brwqg cvqzv
ivzkp noult spwcd

In the order of last_name_with_comma_first:
nqzlq, mbpjs yimpg
etrtd, eyzwe jdizj
vrnup, xkdkg cpltr
irxza, obamv qzcwu
kkghz, thwsz pxche
cqjwk, rcprd okdjr
zscmh, rtjtk grozs
```

```
gtpve, jfvcy fbice
wmihs, brwqg cvqzv
ivzkp, noult spwcd

nqzlq
etrtd
vrnup
irxza
kkghz
cqjwk
zscmh
gtpve
wmihs
ivzkp
```

5. Extend the **People** class into a subclass named **PeopleWithMoney**. Endow this class with its own instance variable named **wealth**. Initialize wealth with 10 randomly generated integer between 0 and 1000. Again create an iterator for this class that gets a part of its work done by the iterator of the parent class.
   - When iterating through an instance of **PeopleWithMoney**, that should print each individual's name following the order "first middle last" and the wealth associated with the individual
   - Also make the instances of the subclass callable. When an instance of **PeopleWithMoney** with the function-call operator "()" is called, the names of all individual sorted by the size of their wealth in the ascending order should be printed out

[16]:
```
class PeopleWithMoney(People):
    def __init__(self, nam1, nam2, nam3, order, wealth):
        super().__init__(nam1, nam2, nam3, order)
        self.wealth = wealth

    def __iter__(self):
        return MoneyIterator(self)

    def __call__(self):
        index_order = []
        n = len(self.first_names)
        sorted_name = []
        for ii in range(n):
            index_order = index_order + [ii]
        for ii in range(n):
            min = ii
            for jj in range(ii + 1, n):
                if self.wealth[index_order[jj]] < self.wealth[
                        index_order[min]]:
                    min = jj
            index_order[min], index_order[ii] = index_order[ii], index_order[
```

```
                min]
        for ii in range(n):
            N = index_order[ii]
            sorted_list = [
                self.first_names[N], self.middle_names[N], self.last_names[N],
                str(self.wealth[N])
            ]
            sorted_str = " ".join(sorted_list)
            sorted_name.append(sorted_str)
            if ii < n - 1:
                print(sorted_str, end="\n")
            else:
                print(sorted_str, end="")
        return sorted_name


class MoneyIterator(PeopleIterator):
    def __init__(self, MoneyObject):
        super().__init__(MoneyObject)
        self.money = MoneyObject.wealth

    def __iter__(self):
        return self

    def __next__(self):
        Name = super().__next__()
        NameMoneyList = [Name, str(self.money[self.index])]
        NameWithMoney = " ".join(NameMoneyList)
        return NameWithMoney
```

[17]:
```
WealthList = []
for ii in range(10):
    WealthList.append(random.randint(0, 1000))
people4 = PeopleWithMoney(FirstList, MiddleList, LastList, 'first_name_first',
                          WealthList)
for item in people4:
    print(item, end='\n')
print("")
people4()
```

```
mbpjs yimpg nqzlq 601
eyzwe jdizj etrtd 645
xkdkg cpltr vrnup 343
obamv qzcwu irxza 865
thwsz pxche kkghz 194
rcprd okdjr cqjwk 248
rtjtk grozs zscmh 16
jfvcy fbice gtpve 749
```

```
brwqg cvqzv wmihs 277
noult spwcd ivzkp 119

rtjtk grozs zscmh 16
noult spwcd ivzkp 119
thwsz pxche kkghz 194
rcprd okdjr cqjwk 248
brwqg cvqzv wmihs 277
xkdkg cpltr vrnup 343
mbpjs yimpg nqzlq 601
eyzwe jdizj etrtd 645
jfvcy fbice gtpve 749
obamv qzcwu irxza 865
```

[17]:
```
['rtjtk grozs zscmh 16',
 'noult spwcd ivzkp 119',
 'thwsz pxche kkghz 194',
 'rcprd okdjr cqjwk 248',
 'brwqg cvqzv wmihs 277',
 'xkdkg cpltr vrnup 343',
 'mbpjs yimpg nqzlq 601',
 'eyzwe jdizj etrtd 645',
 'jfvcy fbice gtpve 749',
 'obamv qzcwu irxza 865']
```