Multiparadigm Programming with Python Chapter 7

H. Conrad Cunningham

05 April 2022

Contents

7 Python Object Orientation **2**
7.1 Chapter Introduction 2
7.2 Class and Instance Attributes 2
7.3 Object Dictionaries 5
7.4 Special Methods and Operator Overloading 5
7.5 Object Orientation 8
7.5.1 Inheritance 8
7.5.1.1 Understanding relationships among classes . . . 11
7.5.1.2 Replacement and refinement 13
7.5.2 Subtype polymorphism 13
7.5.3 Multiple Inheritance 14
7.6 What Next? 14
7.7 Chapter Source Code 15
7.8 Exercises 15
7.9 Acknowledgements 15
7.10 Terms and Concepts 15
7.11 References 15

Copyright (C) 2018, 2022, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

7 Python Object Orientation

7.1 Chapter Introduction

Chapter 6 examined the basic building blocks of Python programs—statements, functions, classes, and modules.

This chapter (7) examines classes, objects, and object orientation in more detail.

TODO: Chapter goals.

Note: In this book, we use the term Python to mean Python 3. The various examples use Python 3.7 or later.

7.2 Class and Instance Attributes

As we saw in Chapter 6, classes are objects. The class objects can have attributes. Instances of the class are also objects with their own attributes.

Consider the following class `Dummy` which has a class-level variable `r`. This attribute exists even if no instance has been created.

Instances of `Dummy` have instance variables `s` and `t` and an instance method `in_meth`.

```
class Dummy:
    r = 1
    def __init__(self, s, t):
        self.s = s
        self.t = t
   def in_meth(self):
        print('In instance method in_meth')
```

Now consider the following Python REPL session with the above definition.

```
>>> Dummy.r
1
>>> d = Dummy(2,3)
>>> d.s
2
>>> d.in_meth()
>>> In instance method method
```

In the above, we see that:

- `Dummy.r` accesses the value of class variable `r` of the class object for the class `Dummy`.
- `d.s` accesses the value of instance variable `s` of an instance object created by the constructor call and assignment `d = Dummy(2)`.
- `d.in_meth()` calls instance method `in_meth` of instance object `d`.

These usages are similar to those of other object-oriented languages such as Java.

A class can have three different kinds of methods in Python [10]:

1. An *instance method* is a function associated with an instance of the class. It requires a reference to an instance object to be passed as the first non-optional argument, which is by convention named `self`. If that reference is missing, the call results in a `TypeError`.

   It can access the values of any of the instance's attributes (via the `self` argument) as well as the class's attributes.

   Note `in_meth` in the `Dummy` code below.

2. A *class method* is a function associated with a class object. It requires a reference to the class object to be passed as the first non-optional argument, which is by convention named `cls`. If that reference is missing, the call results in a `TypeError`.

   It can access the values of any of the class's attributes (via the `cls` argument). For example, `cls()` can create a new instance of the class. However, it cannot access the attributes of any of the class's instances.

   Note `cl_meth` in the `Dummy` code below.

   Class methods can be overriden in subclasses.

   Because Python does not support method overloading, class methods are useful in circumstances where overloading might be used in a language like Java. For example, we can use class methods to implement factory methods as alternative constructors for instances of the class.

3. A *static method* is a function associated with the class object, but it does not require any non-optional argument to be passed

   A static method is just a function attached to the class's namespace. It cannot access any of the attributes of the class or instances except in a way that any function in the program can (e.g., by using the name of the class explicitly, by being passed an object as an argument, etc.)

   Note `s_meth` in the `Dummy` code below.

   Static methods cannot be overrides in subclasses.

```
class Dummy:  # extended definition
    r = 1

    def __init__(self, s, t):
        self.s = s
        self.t = t

    def in_meth(self):
        print('In instance method in_meth')
```

```python
    @classmethod
    def cl_meth(cls):
        print(f'In class method cl_meth for {cls}')

    @staticmethod
    def st_meth():
        print('In static method st_meth')
```

In the example, the *decorators* `@classmethod` and `@staticmethod` transform the attached functions into class and static methods, respectively. We discuss decorators in Chapter 9.

Now consider a Python REPL session with the extended definition.

```python
>>> d = Dummy(2,3)
>>> d.in_meth()
In instance method in_meth
>>> d.r
1
>>> Dummy.cl_meth()
In class method cl_meth for Dummy
>>> Dummy.st_meth()
In static method st_meth
>>> Dummy.in_meth()
Traceback (most recent call last):
...
TypeError: in_meth() missing 1 required positional argument:
    'self'
>>> type(d.in_meth)
<class 'method'>
>>> type(Dummy.cl_meth)
<class 'method'>
>>> type(Dummy.st_meth)
<class 'function'>
>>> din = d.in_meth  # get method obj, store in var din
>>> din()            # call method object in din
In instance method in_meth
None
>>> type(din)
<class 'method'>
```

Note that the types of the references `d.in_meth` and `Dummy.cl_meth` are both `method`. A `method` object is essentially a function that binds in a reference to the required first positional argument. A method object is, of course, a first-class object that can be stored and invoked later as illustrated by variable `din` above.

However, note that `Dummy.st_meth` has type `function`.

The Python 3.7+ source code for the class `Dummy` is available in file `dummy1,py`.

7.3 Object Dictionaries

As we noted in Chapter 5, each Python object has a distinct dictionary that maps the local names to the data attributes and operations (i.e., its environment). Each object's attribute `__dict__` holds its dictionary. Python programs can access this dictionary directly.

Again consider the `Dummy` class we examined in Section 7.2. Let's look at dictionary for this class and an instance in the Python REPL.

```
>>> d = Dummy(2,3)
>>> d.__dict__
{'s': 2, 't': 3}
>>> Dummy.__dict__["r"]
1
>>> Dummy.__dict__["in_meth"]
<function Dummy.in_meth at 0x10191abf8>
>>> Dummy.__dict__["cl_meth"]
<classmethod object at 0x101928c50>
>>> Dummy.__dict__["st_meth"]
<staticmethod object at 0x101928c88>
```

TODO: Investigate and explain last two types returned above?

7.4 Special Methods and Operator Overloading

Almost everything about the behavior of Python classes and instances can be customized. A key way to do this is by defining or redefining *special methods* (sometimes called *magic* methods).

Python uses special methods to provide an *operator overloading* capability. There are special methods associated with certain operations that are invoked by builtin operators (such as arithmetic and comparison operators, subscripting) and with other functionality (such as initializing newly class instance).

The names of special methods both begin and end with double underscores `__` (and thus are sometimes called "dunder" methods). For example, in an earlier subsection, we defined the special method `__init__` to specify how a newly created instance is initialized. In other class-based examples, we have defined the `__str__` special method to implement a custom string conversion for an instance's state.

Consider the class `Dum` that overrides the definition of the addition operator to do the same operation as multiplication.

```
class Dum:
    def __init__(self,x):
```

```
        self.x = x
    def __add__(a,b):
        return a.x * b.x
```

Now let's see how `Dum` works.

```
>>> y = Dum(2)
>>> z = Dum(4)
>>> y + z
8
```

Consider the rudimentary `SparseArray` collection below. It uses the special methods `__init__`, `__str__`, `__getitem__`, `__setitem__`, `__delitem__`, and `__contains__` to tie this new collection into the standard access mechanisms. (This example stores the sparse array in a dictionary internally, but "hides" that from the user.)

The Boolean `__contains__` functionality searches the `SparseArray` instance for an item. The class also provides a separate Boolean method `has_index` to check wether an index has a corresponding value. Alternatively, we could have tied the `__contains__` functionality to the latter and provided a `has_item` method for the former.

In addition, the method `from_assoc` loads an "association list" into a sparse array instance. Here, the term *association list* refers to any iterable object yielding a finite sequence of index-value pairs.

Similarly, the method `to_assoc` unloads the entire sparse array into a sorted list of index-value pairs (which is an iterable object).

For simplicity, the implementation below just prints error messages. It probably should raise exception instead.

```
class SparseArray:

    def __init__(self, assoc=None):
        self._arr = {}
        if assoc is not None:
            self.from_assoc(assoc)

    def from_assoc(self,assoc):
        for p in assoc:
            if len(p) == 2:
                (i,v) = p
                if type(i) is int:
                    self._arr[i] = v
                else:
                    print(
                        f'Index not int in assoc list: {str(i)}')
            else:
```

```python
                print(
                    f'Invalid pair in assoc list: {str(p)}')

    def has_index(self, index):
        if type(index) is int:
            return index in self._arr
        else:
            print(
                f'Warning: Index not int: {index}')
            return False

    def __getitem__(self, index):         #arr[index]
        if type(index) is int:
            return self._arr[index]
        else:
            print(f'Index not int: {index}')

    def __setitem__(self, index, value): #arr[index]=value
        if type(index) is int:
            self._arr[index] = value
        else:
            print(f'Index not int: {index}')

    def __delitem__(self, index):         #del arr[index]
        if type(index) is int:
            del self._arr[index]
        else:
            print(f'Index not int: {index}')

    def __contains__(self, item):         #item value in arr
        return item in self._arr.values()

    def to_assoc(self):
        return sorted(self._arr.items())

    def __str__(self):
        return str(self.to_assoc())
```

Now consider a Python REPL session with the above class definition.

```
>>> arr = SparseArray()
>>> type(arr)
<class '__main__.SparseArray'>
>>> arr
[]
>>> arr[1] = "one"
>>> arr
```

```
[(1, `one`)]
>>> arr.has_index(1)
True
>>> arr.has_index(2)
False
>>> arr.from_assoc([(2,"two"),(3,"three")])
{1: 'one', 2: 'two', 3: 'three'}
>>>
>>> arr[10] = "ten"
>>> arr
[(1,'one'), (2, 'two'), (3, 'three'), (10, 'ten')]
>>> del arr[3]
>>> arr
[(1,'one'), (2, 'two'), (10, 'ten')]
>>> 'ten' in arr
True
```

The Python 3.7+ source code for the class `SparseArray` is available in file `sparse_arrat1,py`.

7.5 Object Orientation

TODO: Remove any unnecessary duplication among this discussion and similar discussions in Chapters 3 and 5.

Chapter 3, Object-Based Paradigms, of *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [6] discusses object orientation in terms of a general object model. The general *object model* includes four basic components:

1. objects
2. classes
3. inheritance
4. subtype polymorph

We discuss Python's objects in Chapter 5 and classes in Chapter 6.

Now let's consider the other two components of the general object model in relation to Python.

7.5.1 Inheritance

In programming languages in general, inheritance involves defining hierarchical relationships among classes. From a *pure* perspective, a class `C` *inherits* from class `P` if `C`'s objects form a *subset* of `P`'s objects in the following sense:

- Class `C`'s objects must support all of class `P`'s operations (but perhaps are carried out in a special way).

  We can say that a `C` object *is a* `P` object or that a class `C` object can be *substituted* for a class `P` object whenever the latter is required.

- Class `C` may support additional operations and an extended state (i.e., more data attributes fields).

We use the following terminology.

- Class `C` is called a *subclass* or a *child* or *derived class*.
- Class `P` is called a *superclass* or a *parent* or *base* class.
- Class `P` is sometimes called a *generalization* of class `C`; class `C` is a *specialization* of class `P`.

In terms of the discussion in the Type System Concepts section, the parent class `P` defines a conceptual type and child class `C` defines a behavioral subtype of `P`'s type. The subtype satisfies the *Liskov Substitution Principle* [8,11].

Even in a statically typed language like Java, the language does not enforce this subtype relationship. It is possible to create subclasses that are not subtypes. However, using inheritance to define subtype relationships is considered good object-oriented programming practice in most circumstances.

In a dynamically typed like Python, there are fewer supports than in statically typed languages. But using classes to define subtype relationships is still a good practice.

The importance of inheritance is that it encourages sharing and reuse of both design information and program code. The shared state and operations can be described and implemented in parent classes and shared among the child classes.

The following code fragment shows how to define a single inheritance relationship among classes in Python. Instance method `process` is defined in the parent class `P` and *overridden* (i.e., redefined) in child class `C` but not overriden in child class `D`. In turn, `C` 's instance method `process` is overridden in its child class `G`.

```
class P:
    def __init__(self,name=None):
        self.name = name
    def process(self):
        return f'Process at parent P level'

class C(P):   # class C inherits from class P
    def process(self):
        result = f'Process at child C level'
        # Call method in parent class
        return f'{result} \n  {super().process()}'

class D(P):   # class D inherits from class P
    pass

class G(C):   # class G inherits from class C
```

```
    def process(self):
        return f'Process at grandchild G level'
```

Now consider a (lengthy) Python REPL session with the above class definition.

```
>>> p1 = P()
>>> c1 = C()
>>> d1 = D()
>>> g1 = G()
>>> p1.process()
'Process at parent P level'
>>> c1.process()
'Process at child C level'
'Process at parent P level'
>>> d1.process()
'Process at parent P level'
>>> g1.process()
'Process at grandchild G level'
#
>>> type(P)
<class 'type'>
>>> type(C)
<class 'type'>
>>> type(G)
<class 'type'>
>>> issubclass(P,object)
True
>>> issubclass(C,P)
True
>>> issubclass(G,C)
True
>>> issubclass(G,P)
True
>>> issubclass(G,object)
True
>>> issubclass(C,G)
False
>>> issubclass(G,D)
False
>>> issubclass(P,type)
False
>>> isinstance(P,type)
True
>>> isinstance(C,type)
True
>>> isinstance(G,type)
True
```

```
>>> type(type)
<class 'type'>
>>> issubclass(type,object)
True
>>> isinstance(type,type)
True
>>> type(object)
<class 'type'>
>>> isinstance(object,type)
True
>>> issubclass(object,type)
False
```

Note: The Python 3.7+ source code for the above version of the `P` class hierarchy is available in file `inherit1.py`.

7.5.1.1 Understanding relationships among classes By examining the REPL session above, we can observe the following:

- Top-level user-defined classes like `P` implicitly inherit from the `object` root class. They have the `issubclass` relationship with `object`.
- A user-defined subclass like `C` inherits explicitly from its superclass `P`, which inherits implicitly from root class `object`. Class `C` thus has `issubclass` relationships with both `P` and `object`.
- By default, all Python classes (including subclasses) are instances of the root metaclass `type` (or one of its subtypes as we see later). But non-class objects are not instances of `type`.

As we noted in Chapter 6, we call class objects *metaobjects*; they are constructors for ordinary objects [1,7].

Also as we noted in Chapter 6, we call special class objects like `type` *metaclasses*; they are constructors for metaobjects (i.e., class objects) [1,7].

Note that classes `object` and `type` have special – almost "magical" – relationships with one another [9:593–595].

- Class `object` is an instance of class `type` (i.e., it is a Python class object).
- Class `type` is an instance of itself (i.e., it is a Python class object) and a subclass of class `object`.

The diagram in Figure 7.1 shows the relationships among user-definfed class `P` and built-in classes `int`, `object`, and `type`. Solid lines denote subclass relationships; dashed lines denote "instance of" relationships.

7.5.1.2 Replacement and refinement There are two general approaches for overriding methods in subclasses [4]:

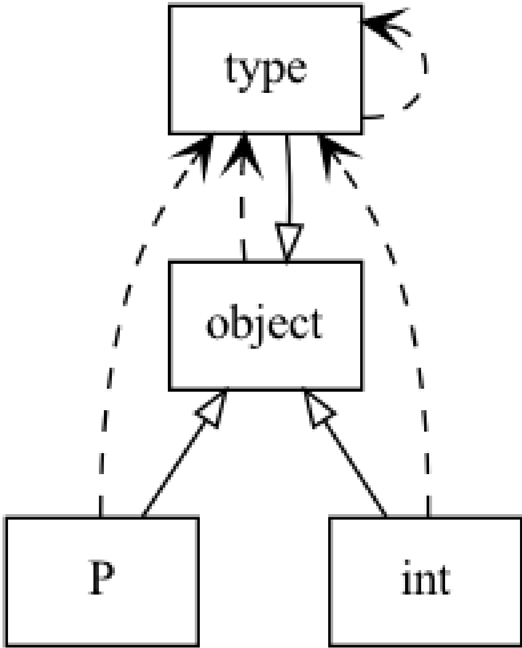

Figure 7.1: Python Class Model

- *Replacement*, in which the child class method totally replaces the parent class method

  This is the usual approach in most "American school" object-oriented languages in use today—Smalltalk (where it originated), Java, C++, C#, and Python.

- *Refinement*, in which the language merges the behaviors of the parent and child classes to form a new behavior

  This is the approach taken in Simula 67 (the first object-oriented language) and its successors in the "Scandinavian school" of object-oriented languages. In these languages, the child class method typically wraps around a call to the parent class method.

The refinement approach supports the implementation of pure subtyping relationships better than replacement does. The replacement approach is more flexible than refinement.

A language that takes the replacement approach usually provides a mechanism for using refinement. For example in the Python class hierarchy example above, the expression `super().process()` in subclass `C` calls the `process` method of its superclass `P`.

7.5.2 Subtype polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. As we saw in Chapter 5, polymorphism appears in several forms in programming languages. Here we examine one form.

In the Python class hierarchy example above, the method `process` forms part of the common interface for this hierarchy. Parent class `P` defines the method, child class `C` overrides `P` 's definition by refinement, and grandchild class `G` overrides `C` 's definition by replacement. However, child class `D` does not override `P` 's definition.

Subtype polymorphism (sometimes called *polymorphism by inheritance*, *inclusion polymorphism*, or *subtyping*) means the association of an operation invocation (e.g., method call) with the appropriate operation implementation in an inheritance (i.e., subtype) hierarchy.

This form of polymorphism is usually carried out at run time. Such an implementation is called *dynamic binding*.

In general, given an object (i.e., class instance) to which an operation is applied, the runtime system first searches for an implementation of the operation associated with the object's class. If no implementation is found, the system checks the parent class, and so forth up the hierarchy until it finds an implementation

and then invokes it. Implementations of the operation may appear at several levels of the hierarchy.

The combination of dynamic binding with a well-chosen inheritance hierarchy allows the possibility of an instance of one subclass being substituted for an instance of a different subclass during execution. Of course, this can only be done when none of the extended operations of the subclass are being used.

In a statically typed language like Java, we declare a variable of some ancestor class type. We can then store any descendant class instance in that variable. Polymorphism allows the program to apply any of the ancestor class operations to the instance.

Because of dynamically typed variables, polymorphism is even more flexible in Python than in Java.

In Python, an instance object may also have its own implementation of a method, so the runtime system searches the instance before searching upward in the class hierarchy.

Also (as we noted in an earlier section) Python uses duck typing. Objects can have a common interface even if they do not have common ancestors in a class hierarchy. If the runtime system can find an compatible operation associated with an instance, it can execute it.

Thus Python's approach to subtype polymorphism gives considerable flexibility in structuring programs. However, unlike statically typed languages, the compiler provides little help in ensuring the compatibility of method implementations.

Again consider the simple inheritance hierarchy above in the following Python REPL session.

```
>>> d1 = D()
>>> g1 = G()
>>> obj = d1     # variables support polymorphism
>>> obj.process()
'Process at parent P level'
>>> obj = g1     # variables support polymorphism
>>> obj.process()
'Process at grandchild G level'
```

7.5.3 Multiple Inheritance

TODO: Discuss multiple inheritance. Issues include the diamond problem, Python syntax and semantics, and method resolution order.

7.6 What Next?

TODO

7.7 Chapter Source Code

TODO

7.8 Exercises

TODO

7.9 Acknowledgements

In Spring 2018, I drafted what is now this chapter as part of the document Basic Features Supporting Metaprogramming, which is Chapter 2 of the 3 chapters of the booklet *Python 3 Reflexive Metaprogramming* [5]. The Spring 2018 material used Python 3.6.

The overall booklet *Python 3 Reflexive Metaprogramming* is inspired by David Beazley's Python 3 Metaprogramming tutorial slides from PyCon'2013 [2]. In particular, I adapted and extended the "Basic Features" material from the terse introductory section of Beazley's tutorial; I attempted to answer questions I had as a person new to Python. Beazley's tutorial draws on material from his and Brian K. Jones' book *Python Cookbook* [3].

In Fall 2018, I divided the Basic Features Supporting Metaprogramming document into 3 chapters—Python Types, Python Program Components, and Python Object Orientation (this chapter). I then revised and expanded each. These 2018 chapers use Python 3.7.

This chapter seeks to be compatible with the concepts, terminology, and approach of my textbook *Exploring Languages with Interpreters and Functional Programming* [6], in particular of Chapters 2, 3, 5, 6, 7, 11, and 21.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

7.10 Terms and Concepts

TODO

7.11 References

[1] 1991. *The art of the metaobject protocol.* MIT Press, Cambridge, Massachusetts, USA.

[2] David Beazley. 2013. Python 3 metaprogramming (tutorial). Retrieved from http://www.dabeaz.com/py3meta/

[3] David Beazley and Brian K. Jones. 2013. *Python cookbook* (Third ed.). O'Reilly Media, Sebastopol, California, USA.

[4] Timothy Budd. 2002. *An introduction to object oriented programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA. Retrieved from https://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/toc.pdf

[5] H. Conrad Cunningham. 2018. Python 3 reflexive metaprogramming. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci658/notes/PythonMetaprogramming/Py3RefMeta.html

[6] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[7] Ira R. Forman and Scott Danforth. 1999. *Putting metaclasses to work: A new dimension in object-oriented programming.* Addison-Wesley, Boston Massachusetts, USA.

[8] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[9] Luciano Ramalho. 2013. *Fluent Python: Clear, concise, and effective programming.* O'Reilly Media, Sebastopol, California, USA.

[10] StackOverflow. 2012. Meaning of @classmethod and @staticmethod for beginner? Retrieved from https://stackoverflow.com/questions/12179271/meaning-of-classmethod-and-staticmethod-for-beginner

[11] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle