

# Data types as objects

## Chapter 17

# This chapter covers

- Treating types as objects
- Using types
- Creating user-defined classes
- Understanding duck typing
- Using special method attributes
- Subclassing built-in types

# Types are objects, too

- This example is the first time you've seen the built-in type function in Python.
- It can be applied to any Python object and returns the type of that object.
- In this example, the function tells you that 5 is an int (integer) and that ['hello', 'goodbye'] is a list—things that you probably already knew.

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

# Types are objects, too

- The object returned by `type` is an object whose type happens to be `<class 'type'>`; you can call it a type object.
- A type object is another kind of Python object whose only outstanding feature is the confusion that its name sometime causes.

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

# Using types

- Now that you know that data types can be represented as Python type objects, what can you do with them?
- You can compare them, because any two Python objects can be compared.

```
>>> type("Hello") == type("Goodbye")
True
>>> type("Hello") == type(5)
False
```

```
>>> class C:
...     pass
...
>>> class D:
...     pass
...
>>> class E(D):
...     pass
...
>>> x = 12
>>> c = C()
>>> d = D()
>>> e = E()
>>> isinstance(x, E)
False
>>> isinstance(c, E)
False
>>> isinstance(e, E)
True
>>> isinstance(e, D)
True
>>> isinstance(d, E)
False
>>> y = 12
>>> isinstance(y, type(5))
True
```

# Types and user-defined classes

- The most common reason to be interested in the types of objects, particularly
- instances of user-defined classes, is to find out whether a particular object is an instance of a class.
- After determining that an object is of a particular type, the code can treat it appropriately.

# Duck typing

- Using `type`, `isinstance`, and `issubclass` makes it fairly easy to make code correctly determine an object's or class's inheritance hierarchy.
- Although this process is easy, Python also has a feature that makes using objects even easier: duck typing.
- Duck typing (as in "If it walks like a duck and quacks like a duck, it probably is a duck") refers to Python's way of determining whether an object is the required type for an operation, focusing on an object's interface rather than its type.
- If an operation needs an iterator, for example, the object used doesn't need to be a subclass of any particular iterator or of any iterator at all.
- Duck typing can increase the flexibility of well-written code and, combined with the more advanced object-oriented features, gives you the ability to create classes and objects to cover almost any situation.

# What is a special method attribute?

- A special method attribute is an attribute of a Python class with a special meaning to Python.
- It's defined as a method but isn't intended to be used directly as such.
- Special methods aren't usually directly invoked; instead, they're called automatically by Python in response to a demand made on an object of that class.
- Perhaps the simplest example is the `__str__` special method attribute.
- If it's defined in a class, any time an instance of that class is used where Python requires a user-readable string representation of that instance, the `__str__` method attribute is
- invoked, and the value it returns is used as the required string.



# Example

```
1  ✓ class Color:
2  ✓     def __init__(self, red, green, blue):
3      |         self._red = red
4      |         self._green = green
5      |         self._blue = blue
6  ✓     def __str__(self):
7      |         return "Color: R={0:d}, G={1:d}, B={2:d}".format (self._red, self._green, self._blue)
```

```
>>> c = Color(15, 35, 3)
>>> print(c)
Color: R=15, G=35, B=3
```

# Subclassing from built-in types

- Instead of creating a class for a typed list from scratch, as you did in the previous examples, you can subclass the list type and override all the methods that need to be aware of the allowed type.
- One big advantage of this approach is that your class has default versions of all list operations because it's a list already.
- The main thing to keep in mind is that every type in Python is a class, and if you need a variation on the behavior of a built-in type, you may want to consider subclassing that type.

```
1  class TypedListList(list):
2      def __init__(self, example_element, initial_list=[]):
3          self.type = type(example_element)
4          if not isinstance(initial_list, list):
5              raise TypeError("Second argument of TypedList must "
6                              "be a list.")
7          for element in initial_list:
8              self.__check(element)
9          super().__init__(initial_list)
10
11     def __check(self, element):
12         if type(element) != self.type:
13             raise TypeError("Attempted to add an element of "
14                             "incorrect type to a typed list.")
15     def __setitem__(self, i, element):
16         self.__check(element)
17         super().__setitem__(i, element)
```

# Summary

- Python has the tools to do type checking as needed in your code, but by taking advantage of duck typing, you can write more flexible code that doesn't need to be as concerned with type checking.
- Special method attributes and subclassing built-in classes can be used to add list-like behavior to user-created classes.
- Python's use of duck typing, special method attributes, and subclassing makes it possible to construct and combine classes in a variety of ways..