PV248 Python

Petr Ročkai

Part 1: Object Model

In this lecture, we will look at the role of objects and their semantics. In particular, we will look at the basic structure of a Python program and note that pretty much everything is an object.

Objects

3

- the basic 'unit' of OOP
- they bundle data and behaviour
- provide encapsulation
- make code re-use easier
- also known as 'instances'

The traditional notion of an object, as an entity that encapsulates both data and behaviour, holds quite well in Python. Let's, for a moment, forget the standard story about how the data is enclosed in a wall made of methods (which is not really helpful). This does not work in Python anyway, since attributes (data) are public.
The real benefit is that holding data in discrete packages, accompanied by the relevant chunks of functionality, allows us to think in terms of local invariants. A typical object wraps up a chunk of state, but of course not all combinations of values in its attributes need to be valid (and rarely are, for non-trivial objects). The constraints which govern valid states are what we call (local) invariants.
The methods (behaviour) are then written in such a way that they move the object from valid states to other valid states: in other words, they preserve invariants. In a correctly designed API, it is should be impossible to get an object into an invalid state by only interacting with it through its public interface.

Classes

4

- templates for objects (`class Foo: pass`)
- each (Python) object belongs to a class
- classes themselves are also objects
- calling a class creates an instance
  - `my_foo = Foo()`

Like with plain old data, where values are classified into data types, it is often advantageous to classify objects (which are a kind of value) into classes (which are a kind of data type).
Like any other data type, a class prescribes the structure of the data (attributes) and operations on its values (methods). The distinct feature of class-based object-oriented languages is the ease with which we can create new data types with complex, user-defined operations.
Aside: in Python-speak, since double underscores are so common, they have a special name: 'dunder'. We will call them that in this subject, since we will encounter quite a few of them.
Syntactically, classes in Python behave like functions, which create new instances of the class when called. In addition to creating a new empty object, it will also immediately call the `__init__` method of that object, forwarding any arguments passed to the original call.

Types vs Objects

5

- class system is a type system
- 'duck typing': quacks, walks like a duck
- since Python 3, types are classes
- everything is dynamic in Python
  - you can create new classes at runtime
  - you can pass classes as function parameters

In fact, all data types in Python are actually classes. Each object dynamically belongs to a class: in Python, variables do not carry types (i.e. each variable in Python can hold a value of any type). However, values (objects) do, and it is impossible to accidentally coerce an object into a different class – we say that Python has a strong dynamic type system.

Poking at Classes

6

- `{}.__class__`
- `{}.__class__.__class__`
- `(0).__class__`
- `[].__class__`
- compare `type(0)`, etc.

- `n = numbers.Number(); n.__class__`

Of course, since this is Python, classes also exist at runtime: and as we will eventually learn, everything in Python that exists at runtime is an object. Hence, classes are also objects with attributes and methods and it is possible to interact with them like with any other object. This is not directly useful, but it does serve to illustrate a point about Python. To demonstrate the point about classes being objects, here are some examples of valid Python expressions. Try them yourself and think about what this means.

Encapsulation

7

- objects hide implementation details
- classic types structure data
  - objects also structure behaviour
- facilitates loose coupling

While strictly speaking, there is no encapsulation in Python (hiding of data is not enforced), it is commonly approximated: attributes can be named with a single leading underscore to signify they are 'private' and directly accessing those outside of the class itself is frowned upon. But there is no enforcement mechanism.
Encapsulation is really just a way to structure your code to make local invariants (as mentioned earlier) easier to reason about. All code that could possibly violate a local constraint is bundled up within a class. If some outside code changes the data (attributes) directly and breaks the invariant, it is going to get exactly what it asked for.
The partitioning of program state that is achieved is also known as loose coupling.

Loose Coupling

8

- coupling is a degree of interdependence
- more coupling makes things harder to change
  - it also makes reasoning harder
- good programs are loosely coupled
- cf. modularity, composability

Coupling is a measure of how tangled the program is – how many things break when you change one thing, and how far away from the change. Consider a ticket booking system that started out as a small local thing, with only a few venues. But the system grew and now you are adding venues from another city – but for that, you need to change how venues are stored, since you did not have a 'city' field (they were all local).
In a loosely coupled system, you add the attribute (or a database column) and that's pretty much it: you may need to adjust a few methods of the venue class, but outside of the class, nothing needs to change. In a tightly coupled system, on the other hand, it could easily turn out that now the reservation emails are completely jumbled. This might sound like a stretch, but consider a system which stores all the attributes in a list, and each component simply hard-codes the meaning of each index. So `reservation[4]` is the venue name, and `reservation[7]` is the customer's family name. But you add the city, say as index 5, and now all other fields have shifted, and your entire application is completely broken and you are sending out mails that start 'Dear Britney Spears'. Of course, this is terrible programming and the example is slightly artificial, but things like this do happen.

Polymorphism

9

- objects are (at least in Python) polymorphic
- different implementation, same interface
- only the interface matters for composition
- facilitates genericity and code re-use
- cf. 'duck typing'

One of the principles that helps to achieve loosely coupled programs is polymorphism, where different objects can fill the same role. This encourages loose coupling, because polymorphic objects separate the interface from the implementation quite strictly. If a function must be able to work with objects of different types, it can only do so through a fairly well-defined interface.
One area where polymorphism is very important is generic functions: consider `len`, which is a builtin Python function that gives you a number of elements in a collection. It works equally well for lists, sets, dictionaries and so on. Or consider the `for` loop, that likewise works for multiple collection types. Or the addition operator: you can add integers, floating point numbers, but also strings or lists. Those are all different manifestations of polymorphism.
We have already mentioned that Python is a very dynamic language (and this will come up many more times in the future). Where polymorphism is concerned, this dynamic nature translates into duck typing: types are not rigorous entities like in other languages. This essentially says, that for type-checking purposes, if it quacks like a duck and walks like a duck, it may as well be a duck. Hence, a function that expects a duck will accept anything that has sufficiently duck-like behaviour. In particular, it needs to provide the right methods with the right signatures.

Generic Programming

10

- code re-use often saves time
  - not just coding but also debugging
  - re-usable code often couples loosely
- but not everything that can be re-used should be
  - code can be too generic
  - and too hard to read

Programming the same thing over and over again for slightly altered circumstances gets boring very quickly. And it is not a very efficient use of time. A lot of programming boils down to identifying such repetitive patterns and isolating the common parts. Many of the constructs in programming languages exist to allow exactly this:

- loops let us repeat statements over multiple values,
- functions let us execute a set of statements in different contexts (and with diferent values),
- conditionals allow us to write a sequence of statements that is almost the same in different circumstances,
- user-defined data types (records, classes) allow us to give structure to our data once and use it in many places.

Generic programming is a technique, which mainly gives additional power to functions and objects. Under normal circumstances, functions are parametrized by values which are all of the same type. That is, functions take arguments, which have fixed types, but the values that are passed to the function vary from invocation to invocation. With generic programming, the types can also vary. In Python, generic programming is so pervasive that you may fail to even notice it. You can often pass sets or lists into the same function, without even thinking about it. Or integers vs floating-point numbers. And so on. However, this is not a given. This level of flexibility is also part of the reason why Python programs execute slowly.

Attributes

11

- data members of objects
- each instance gets its own copy
- like variables scoped to object lifetime
- they get names and values

Objects can have attributes (and in fact, most objects do). Attributes form the data associated with an object (as opposed to methods, which form the behaviours).
Attributes behave like variables, but instead of a running function, they are associated with an object. Each instance gets their own copy of attributes, just like each instance of a running function gets its own copy of local variables.
Also like variables, attributes have names and those names are bound to values (within each object instance). Alternatively, you can think of attributes like key-value pairs in a dictionary data structure. Incidentally, this is how most objects are actually implemented in Python.

Methods

12

- functions (procedures) tied to objects
- they can access the object (`self`)
- implement the behaviour of the object
- their signatures (usually) provide the interface
- methods are also objects

Methods are basically just functions, with two differences. The first is

semantic: they are associated with the object and are allowed to access and change its internal state. In Python, this is essentially by custom – any function can, in principle, change the internal state of objects. However, in other languages, this is often not the case.
The second is syntactic: the first argument of a method is the object on which it should operate, but when the method is called, this argument is not passed in explicitly. Instead, we write `instance.method()` and this 'dot syntax' does two things:

1. First, it looks up the method to call: different classes (and hence, different objects) can provide different methods under the same name. In this sense, objects and classes work as a namespace.
2. It passes the object on the left side of the dot to the method as its first parameter.

The list of methods, along with the number and (implied) types of their arguments forms the interface of the object. In Python, it is also fairly normal that some attributes are part of the interface. However, this is not as bad as it sounds, because Python can transparently map attribute access (both read and write) to method calls, using the `@property` decorator. This means that exposing attributes is not necessarily a violation of encapsulation and loose coupling. We will talk about this ability in a later lecture.
Finally, as we have mentioned earlier, everything that exists at runtime is an object: methods exist at runtime, and methods are also objects (just like regular functions).

Class and Instance Methods 13

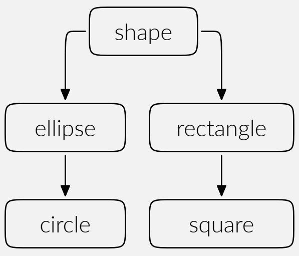
- methods are usually tied to instances
- recall that classes are also objects
- class methods work on the class (`cls`)
- static methods are just namespaced functions
- decorators `@classmethod`, `@staticmethod`

Besides 'standard' methods, which operate on objects (instances), in Python there are two other method types: static, which are really just plain functions hidden inside a class, and class methods which treat the class as an object. They are declared using decorators, like this:

```
@staticmethod
def a_static_method(x, y):
    pass
```

Design-wise, static and class methods are of minor significance.

Inheritance 14



- `class Ellipse( Shape ): ...`
- usually encodes an is-a relationship

In object oriented programs, inheritance plays two roles. The first is organizational: classes are arranged into a hierarchy where each subclass is a more specific version of its superclass. Instances of a subclass are automatically also instances of the superclass. We say that a circle is an ellipse which is a shape. Any interfaces and behaviours common to all shapes are also present on ellipses, circles, rectangles and so on.
The second role is code re-use. While the organizational role is undisputed, using inheritance for code re-use is somewhat controversial. This is because it often, though not always, goes against the former, which is seen as more important.
In the specific picture, maximizing code re-use through inheritance would invert the hierarchy: circle has a single diameter, while an ellipse has two semi-axes. Likewise, a square has one side length, but rectangle has two. The attributes of an ellipse are a superset of the attributes of a circle, and the same with square vs rectangle. Hence, re-use would push us to derive an ellipse from a circle and a rectangle from a square. But not every rectangle is a square – we have saved a bit of code, but ruined the hierarchy.

Multiple Inheritance 15

- more than one base class is possible
- many languages restrict this
- Python allows general M-I
  - `class Bat( Mammal, Winged ): pass`
- 'true' M-I is somewhat rare
  - typical use cases: mixins and interfaces

In both cases (inheritance as an organisational principle, as well as inheritance for code re-use), it sometimes makes sense for a given class to have more than one superclass. In general multiple inheritance, this is allowed without restrictions: a bat inherits both traits of mammals (e.g. that females may be pregnant) and of winged creatures (say, a wingspan). Class hierarchies with this type of multiple inheritance are rare in practice, since they become hard to reason about quite quickly.
There are two types of restricted multiple inheritance: one is mixins, which exist for code re-use, but do not participate in the is-a relationship. The second is interfaces, which is the exact opposite: they provide organization, but no code. In Python, interfaces are often known as Abstract Base Classes.

Mixins 16

- used to pull in implementation
  - not part of the is-a relationship
  - by convention, not enforced by the language
- common bits of functionality
  - e.g. implement `__gt__`, `__eq__` &c. using `__lt__`
  - you only need to implement `__lt__` in your class

Mixins are a form of inheritance (usually multiple inheritance) where only code re-use is a concern. That is, the base class and the derived class are not related conceptually. Some languages have a special 'mixin' construct for this, though more often, this is simply a convention. Programmers understand that they should not use the base type as a general form of all its derived classes. Python belongs to this latter category.

Interfaces 17

- realized as 'abstract' classes in Python
  - just throw a `NotImplemented` exception
  - document the intent in a docstring
- participates in is-a relationships
- partially displaced by duck typing
  - more important in other languages (think Java)

Interfaces are the dual to mixins: they only exist to give structure to the object hierarchy, but do not provide any code sharing. In Python, interfaces are rarely used, because duck typing covers most of the use cases traditionally served by interfaces.
Nonetheless, when an interface is called for in Python, there is no special syntax: by convention, an interface is a class with no attributes and with methods which all throw `NotImplemented`.

Composition

18

- attributes of objects can be other objects
  - (also, everything is an object in Python)
- encodes a has-a relationship
  - a circle has a center and a radius
  - a circle is a shape

Object composition is another important approach to code re-use (ostensibly much more important than inheritance, even though it is often neglected). Where inheritance encodes an is-a relationship, composition encodes a has-a relationship. Since in Python, everything is an object, each attribute is an instance of object composition. In languages with data types other than objects, the distinction is more useful.

Constructors

19

- this is the `__init__` method
- initializes the attributes of the instance
- can call superclass constructors explicitly
  - not called automatically (unlike C++, Java)
  - `MySuperClass.__init__( self )`
  - `super().__init__` (if unambiguous)

We have mentioned local invariants earlier. Object initialization is an important part of that story – the initialization method is responsible for putting the object into a consistent 'blank' or 'base' state. After initialization, all local invariants should hold. Correct methods invoked on consistent objects should keep the object consistent. Together, those two properties mean that correct objects should never violate their local invariants.
In Python, there is a bit of a catch: attributes of superclasses are initialized by the `__init__` method of that superclass, but this method is not called automatically when the object is created. Therefore, you need to always remember to call the `__init__` method of the superclass whenever appropriate.

Class and Object Dictionaries

20

- most objects are basically dictionaries
- try e.g. `foo.__dict__` (for a suitable `foo`)
- saying `foo.x` means `foo.__dict__["x"]`
  - if that fails, `type(foo).__dict__["x"]` follows
  - then superclasses of `type(foo)`, according to MRO

As mentioned earlier, object attributes behave like dictonaries, and in many cases, that's exactly how they are implemented in Python. It is possible to interact with this dictionary directly, through the (magic) attribute `__dict__`. The exceptions are built-in objects like `int`, `str` or `list` and slot-based objects (more on those in a later lecture).
On objects with dictionaries, saying `foo.x` is interpreted to mean `foo.__dict__['x']`. If the attribute (or method!) is not in the object dictionary, it is looked up in the class dictionary (remember that classes are objects too). Then superclasses and so on. The protocol is rather complicated – you can look up the details in the manual.
Remember that in case of methods, in addition to the lookup described above, the dot syntax also binds the first argument of the method to the object on the left of the dot.
It is possible to add new methods and attributes to objects with dictionaries at runtime (by simply adding them through the `__dict__` pseudo-attribute). This is often called 'monkey patching'. This is not possible with objects without dictionaries (though methods can still be added to their classes, which do have dictionaries).

Writing Classes

21

```
class Person:
  def __init__( self, name ):
    self.name = name
  def greet( self ):
    print( "hello " + self.name )

p = Person( "you" )
p.greet()
```

The syntax for creating new classes is fairly straight-forward. Just remember that attributes are always created in the `__init__` method, never by writing them as variables alongside methods, like in most other languages.

Functions

22

- top-level functions/procedures are possible
- they are usually 'scoped' via the module system
- functions are also objects
  - try `print.__class__` (or `type(print)`)
- some functions are built in (`print`, `len`, ...)

While Python is object-based to the extreme, unlike some other languages (e.g. Java), free-standing functions are allowed, and common. Groups of related functions are grouped into modules (possibly along with some classes). Unlike in C, there is no single global namespace for functions. Like everything else in Python, functions are also objects. For instance:

```
def foo():
    print( "hello" )

>>> type( foo )
<class 'function'>
```

A few universally-useful functions are built into the interpreter, like `print` above, or `len`, `hash`, `eval`, `range` and so on. A complete list can be obtained by asking the interpreter, like this:

```
__builtins__.__dict__.keys()
```

However, besides functions, this list includes a number of classes: the standard types (`int`, `dict` and so on) and the standard exception classes (`SyntaxError`, `NameError` and so on). To show off the introspection capabilities of Python, let's get a list of just the built-in functions:

```
for n, o in __builtins__.__dict__.items():
    if type( o ) == type( print ):
        print( n )
```

Remember though: with great power comes great responsibility.

Modules in Python

- modules are just normal `.py` files
- `import` executes a file by name
  - it will look into system-defined locations
  - the search path includes the current directory
  - they typically only define classes & functions
- `import sys` → lets you use `sys.argv`
- `from sys import argv` → you can write just `argv`

In Python, there is no special syntax for creating modules: each source file can be imported as a module. It is customary that modules do not execute any 'top level' code (that is, statements outside of functions or classes), though this is not enforced in any way. There is, however, special syntax for loading modules: the `import` keyword. On `import foo`, Python will look for a file named `foo.py`, first in the current directory and if that fails, in the standard library (which is located in a system-specific location).
When that file is located, it is executed as if it was a normal program, and at the end, all toplevel names that it has defined become available as attributes of the module. That is, if `foo.py` looks like this:

```
def hello():
    print( "hello" )
```

then `bar.py` may look like this:

```
import foo
foo.hello()
```

What this means is that the `import` in `bar.py` locates `foo.py`, executes it, then collects the toplevel definitions (in this case the function `hello`) and makes them available via the 'dot' syntax under the name of the module (`foo.hello`). Behind the scenes, the module is (of course) an object and the toplevel definitions from the file being imported are attached to that object as standard attributes.