14 Write less, code more

In this section I've compiled a few of the more advanced features that I find interesting – they'll help you to write better code.

14.1 Single dispatcher

I often like to say that Python is a good subset of Lisp, and as time passes I find this to be more and more true. Recently I stumbled across the PEP 443, which describes a way to dispatch generic functions in a similar manner to that provided by CLOS, the Common Lisp Object System.

If you're familiar with Lisp, this won't be new to you. The Lisp object system, which is one of the basic components of Common Lisp, provides a good way to define and handle method dispatching. I'll show you how generic methods work in Lisp first – even if only for the pleasure of including Lisp code in a book on Python!

To begin with, let's define a few very simple classes, without any parent classes or attributes

```
(defclass snare-drum ()
  ())

(defclass cymbal ()
  ())
```

```
(defclass stick ()
  ())

(defclass brushes ()
  ())
```

This defines a few classes: `snare-drum`, `symbal`, `stick` and `brushes`, without any parent class nor attribute. These classes compose a drum kit, and we can combine them to play sound. So we define a `play` method that takes two arguments, and returns a sound (as a string).

```
(defgeneric play (instrument accessory)
  (:documentation "Play sound with instrument and accessory."))
```

This only defines a generic method: it isn't attached to any class, and so cannot yet be called. At this stage, we've only informed the object system that the method is generic and can be called with various arguments. Now we'll implement versions of this method that simulate playing our snare-drum.

```
(defmethod play ((instrument snare-drum) (accessory stick))
  "POC!")

(defmethod play ((instrument snare-drum) (accessory brushes))
  "SHHHH!")
```

Now we've defined concrete methods in code. They take two arguments: `instrument`, which is an instance of `snare-drum`; and `accessory`, which is an instance of `stick` or `brushes`.

At this stage, you should see the first major difference between this system and the Python (or similar) object systems: the method isn't tied to any particular class. The methods are **generic**, and any class can implement them.

Let's try it.

```
* (play (make-instance 'snare-drum) (make-instance 'stick))
"POC!"

* (play (make-instance 'snare-drum) (make-instance 'brushes))
"SHHHH!"

* (play (make-instance 'cymbal) (make-instance 'stick))
debugger invoked on a SIMPLE-ERROR in thread
#<THREAD "main thread" RUNNING {1002ADAF23}>:
  There is no applicable method for the generic function
    #<STANDARD-GENERIC-FUNCTION PLAY (2)>
  when called with arguments
    (#<CYMBAL {1002B801D3}> #<STICK {1002B82763}>).

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [RETRY] Retry calling the generic function.
  1: [ABORT] Exit debugger, returning to top level.

((:METHOD NO-APPLICABLE-METHOD (T)) #<STANDARD-GENERIC-FUNCTION PLAY (2)>  ↩
   #<CYMBAL {1002B801D3}> #<STICK {1002B82763}>) [fast-method]
```

As you can see, which function is called depends on the class of the arguments – the object systems **dispatch** the function calls to the right function for us, depending which classes we pass as arguments. If we call `play` with instances that are not known to the object system, an error will be thrown.

Inheritance is also supported and, the (more powerful and less error prone) equivalent of Python's `super()` is available via `(call-next-method)`.

```
(defclass snare-drum () ())
(defclass cymbal () ())

(defclass accessory () ())
(defclass stick (accessory) ())
(defclass brushes (accessory) ())

(defmethod play ((c cymbal) (a accessory))
  "BIIING!")

(defmethod play ((c cymbal) (b brushes))
  (concatenate 'string "SSHHHH!" (call-next-method)))
```

In this example, we define the `stick` and `brushes` classes as subclasses of `accessory`. The `play` method defined will return the sound *BIIING!*, regardless of what kind of accessory instance is used to play the cymbal – except if it's a `brushes` instance; the most precise method is always called. The `(call-next-method)` function is used to call the closest parent method, and in this case that would be the method which returns *"BIIING!"*.

```
* (play (make-instance 'cymbal) (make-instance 'stick))
"BIIING!"

* (play (make-instance 'cymbal) (make-instance 'brushes))
"SSHHHH!BIIING!"
```

Note that CLOS can define specialized methods which apply to only one instance of a class- using the `eql` specializer.

But if you're really curious about the many features CLOS provides, I suggest that you read the brief guide to CLOS by Jeff Dalton as a starter.

Python implements a simpler version of this workflow with the `singledispatch` function, which will is with Python 3.4 as part of the `functools` module. Here's the rough equivalent of the Lisp program above:

```
import functools

class SnareDrum(object): pass
class Cymbal(object): pass
class Stick(object): pass
class Brushes(object): pass

@functools.singledispatch
def play(instrument, accessory):
    raise NotImplementedError("Cannot play these")

@play.register(SnareDrum)
def _(instrument, accessory):
    if isinstance(accessory, Stick):
        return "POC!"
    if isinstance(accessory, Brushes):
        return "SHHHH!"
    raise NotImplementedError("Cannot play these")
```

We define our four classes, and a base `play` function that raises `NotImplementedError`, indicating that by default we don't know what to do. We can then write a specialized version of this function for a specific instrument, the `SnareDrum`. This function checks which accessory type has been passed, and returns the appropriate sound – or raises `NotImplementedError` again if it doesn't recognise the accessory.

If we run the program, it should work as follows:

```
>>> play(SnareDrum(), Stick())
'POC!'
```

```
>>> play(SnareDrum(), Brushes())
'SHHHH!'
>>> play(Cymbal(), Brushes())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "/home/jd/sd.py", line 10, in play
    raise NotImplementedError("Cannot play these")
NotImplementedError: Cannot play these
>>> play(SnareDrum(), Cymbal())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "/home/jd/sd.py", line 18, in _
    raise NotImplementedError("Cannot play these")
NotImplementedError: Cannot play these
```

The `singledispatch` module checks the class of the first argument passed, and calls the appropriate version of the `play` function. For the `object` class, the first-defined version of the function is always the one which is run – so, if our instrument is an instance of a class that we did not register, this base function will be called.

For those eager to try it out, the `singledispatch` function is provided in Python 2.6 to 3.3, through the Python Package Index.

As we saw in the Lisp version of the code, CLOS provides a multiple dispatcher that can dispatch depending on the type of **any of the arguments** defined in the method prototype, not just the first one. Unfortunately, the Python dispatcher is named *singledispatch* for a good reason: it only knows how to dispatch based on the first

argument. Guido van Rossum wrote a short article called multimethod about this subject a few years ago.

In addition, there's no way to call the parent function directly – no equivalent of either `(call-next-method)` from Lisp, or the Python `super()` function. You'll have to use various tricks to bypass this limitation.

To conclude: while I am really glad that Python is heading in this direction, as it's a really powerful way to enhance an object system, it still lacks a lot of the more advanced features that CLOS provides out of the box.

14.2 Context managers

The `with` statement introduced in Python 2.6 is likely to remind old time Lispers of the various `with-*` macros that are often used in the language. Python provides a similar-looking mechanism, with the use of objects which implement the context management protocol.

Objects like those returned by `open` support this protocol; that's why you can write code along these lines:

```
with open("myfile", "r") as f:
   line = f.readline()
```

The object returned by `open` has two methods, one called `__enter__` and one called `__exit__`; these are called at the start of the `with` block and at the end of it, respectively.

A simple implementation of a context object would be:

Example 14.1 Simple implementation of a context object

```
class MyContext(object):
    def __enter__(self):
        pass
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        pass
```

It wouldn't do anything, but is valid.

When do you want to use context managers? The use of context management protocol might be appropriate if you identify the following pattern in your object:

1. Call method A;
2. Execute some code;
3. Call method B.

Where it is expected that a **call to method B** must **always** be done after a **call to A**. The `open` function illustrates this pattern well – in this case, the constructor that opens the file and allocates a file descriptor internally is method A. The `close` method that releases the file descriptor corresponds to method B. Obviously, the `close` function is always meant to be called **after** you instantiate the file object.

The **`contextlib`** standard library provides **`contextmanager`** to ease the implementation of such a mechanism, by relying on a generator to construct the `__enter__` and `__exit__` methods for you. We can use this to implement our simple context manager:

Example 14.2 Simplest usage of `contextlib.contextmanager`

```
import contextlib


@contextlib.contextmanager
def MyContext():
    yield
```

For example, I've been using this design pattern in Ceilometer for the pipeline infrastructure we set up. Basically, a pipeline is a tube into which objects are put, and

from which they are dispatched to various places. The steps to send data this way are as follows:

1. Call the `publish(objects)` method of a pipeline, with your objects as arguments – as many times as you need.
2. Once done, call the `flush()` method to indicate that you're done publishing for now.

Note that if you never call the `flush()` method, your objects will never be sent down the tube; or at least not completely. It can be very easy for a programmer to forget about a `flush()` call, which breaks the program without giving any clues as to what might be wrong.

It's much better if your API provides a context manager object that will not allow the API user to make this mistake. This can be done pretty easily with the following code:

Example 14.3 Using a context manager on a pipeline object

```
import contextlib

class Pipeline(object):
    def _publish(self, objects):
        # Imagine publication code here
        pass

    def _flush(self):
        # Imagine flushing code here
        pass

    @contextlib.contextmanager
    def publisher(self):
```

```
        try:
            yield self._publish
        finally:
            self._flush()
```

Now, when users of our API wants to publish something in our pipeline, they don't have to use `_publish` or `_flush`. They just request a publisher using the eponym function, and uses it.

```
pipeline = Pipeline()
with pipeline.publisher() as publisher:
    publisher([1, 2, 3, 4])
```

When you provide an API like this, there's no place for user error. Always use context managers when you see that it suits the design pattern.

In some contexts, it might be useful to use several context managers at the same time – for example, opening two files at the same time to copy their content:

Example 14.4 Opening two files at the same time

```
with open("file1", "r") as source:
    with open("file2", "w") as destination:
        destination.write(source.read())
```

Remember that the `with` statement supports having multiple arguments – so you should write:

Example 14.5 Opening two files at the same time with one `with` statement

```
with open("file1", "r") as source, open("file2", "w") as destination:
    destination.write(source.read())
```