Three Years of Python 3

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply/ply.html). He is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

First, I'd like to offer a general welcome. This is the first of what I hope will be an ongoing series of articles about all things Python. This year marks the 21st anniversary of Python's first release. Although Python has been around for some time, its popularity has been growing by leaps and bounds—especially if one looks at the soaring attendance at Python conferences worldwide.

I first discovered Python in 1996 when I was still a graduate student. Since then, it's become my tool of choice for, well, just about everything (although I still do a fair bit of C programming for occasional projects involving embedded systems or high performance computing).

In this column I hope to explore a variety of topics related to modern Python software development. This includes advances in the language itself, interesting new add-on libraries, useful programming techniques, and more. No topic is off-limits, although I'll admit that I do have a certain preference for problems involving data analysis and systems programming. In this issue I start by sharing some of my experiences working with Python 3. In future issues, I hope to look at a variety of other topics, including the PyPy project, the state of Python concurrency, and hidden secrets of the standard library.

Python 3

It's hard to believe, but last December marked the three-year anniversary of the first release of Python 3. Judging by the glacial rate of adoption, you might hardly notice. Honestly, most Python programmers (including most of Python's core developers) are still using various versions of Python 2 for "real work." Nevertheless, Python 3 development continues to move along as the language is improved and more and more libraries start to support it. Although you might look at the adoption rate with some dismay, it has always been known by Python insiders that it could take five years or more for a significant number of users to make the switch. Thus, we're still in the middle of that transition.

A few years ago, I presented a tour of Python 3 (see *;login:* April 2009). At that time, it wasn't all rosy. I noted some major problems, such as horrible I/O performance and complications in the bytes/Unicode interface. For the most part, Python 3 was simply a curiosity and something far too experimental to rely on for "real" work. Since then, a lot of problems have been addressed and the implementation improved. However, if you've been sitting on the sidelines, it's still hard to know exactly what Python 3 offers.

About a year ago, I made a conscious decision to write all new projects in Python 3 to get a better sense of working with it in practice and to explore issues involved in porting existing code. So, in this article, I hope to revisit the topic of Python 3, but with a slightly different spin. Rather than simply rehashing a long list of new language features, I thought I would simply focus on the specific parts of Python 3 that have, through experience, actually proven to be rather useful, surprising, or problematic.

Useful Features

There are certain language features unique to Python 3 that I now find myself using with some regularity. Few of these features have been backported, and they are unlikely to appear in any future version of Python 2.

Sequence Unpacking Wildcards

If you have a Python tuple (or other sequence), you can easily unpack it into variables. For example:

```
>>> row = ('ACME',50,91.15)
>>> name, shares, price = row
>>>
```

In Python 2, such unpacking only works if the number of items in the sequence on the right exactly matches the number of storage locations on the left. However, in Python 3, you can introduce a wildcard that will match any number of items and collect them into a list. For example:

```
>>> name, *last = row
>>> last
[50, 91.15]
>>> *first, price = row
>>> first
['ACME', 50]
>>>
```

Although this feature might seem minor, unpacking sequence data comes up quite a bit in the context of working with tabular data: for example, reading data out of databases, reading lines from CSV files, and so forth. Using the wildcard can be a convenient means to write code that only wants to work with some of the fields or with data that has a varying number of columns.

I have also found wildcard unpacking to be a useful technique for treating Python tuples as a kind of prefixed data structure akin to a Lisp S-expression. Here the first element might be some kind of operator, tag, or identifier, while the remaining elements are data. Thus you can write code that processes the data like this:

```
>>> s = ('+',3,4,5)
>>> op, *data = s
>>> op
'+'
>>> data
[3, 4, 5]
>>>
```

In Python 2 you could achieve the same effect by writing `op, data = s[0], s[1:]`. However, the new version is just a bit more elegant, so why not use it?

Keyword-only Function Arguments

Consider the following Python function:

```
def recv(block=True, timeout=None):
       # Receive a message
       ...
```

One issue with using the above function is that subsequent calls can potentially lead to cryptic code where the meaning of the arguments is not entirely clear to someone reading it. For example:

```
msg = recv(False)  # What does False mean?
msg = recv(1,5)  # 1? 5?
Huh? ...
```

To avoid this, you can force keyword arguments by inserting a `*` into the argument signature:

```
def recv(*,block=True, timeout=None):
       # Receive a message
       ...
```

For all arguments after the `*`, users are forced to use keywords when calling:

```
msg = recv(block=False)  # Ok
msg = recv(block=1,timeout=5)  # Ok
msg = recv(1,5)  # Error
```

Although you could achieve a similar effect in Python 2, it was always rather clumsy. For example, this old code implements the same functionality, but with much less elegance:

```
dec recv(**kwargs):
       block = kwargs.pop('block',True)
       timeout = kwargs.pop('timeout',None)
       if kwargs:
          raise TypeError("Unknown argument(s): %s" % list(kwargs))
       ...
```

One of the reasons I like this feature is that it allows you to write functions that have more precise calling signature and less underlying magic related to fiddling around with various forms of `*args` and `**kwargs`. It even plays nice with documentation and IDEs. For instance, if you ask for `help(recv)`, you'll get more descriptive output showing the argument names as opposed to a vague description of `**kwargs`. Believe it or not, this is one of my most-used Python 3–specific language features. It's nice.

Dictionaries, Sets, and Views

One feature of Python 3 that has taken some time to fully appreciate is the better unification of dictionaries, sets, and the newly introduced dictionary "view" objects. Under the covers, sets and dictionaries are implemented in an almost identical manner. Essentially, a set is just a dictionary, but with keys only.

In Python 2, sets always felt kind of bolted on to the rest of the language (somewhat true, as sets weren't actually introduced until Python 2.3). In Python 3, they are much more integrated with all of the other data types. First, there is new syntax for writing a set:

```
>>> fruits = { 'pear', 'apple', 'banana', 'peach' }
>>>
```

Dictionaries now support a different mechanism for working with the keys, values, or key/value pairs. Consider a simple dictionary:

```
>>> a = {
        'x' : 1,
        'y' : 2,
        'z' : 3
      }
>>>
```

If you ask for the dictionary keys, Python 3 gives you a "key-view" object. What's unusual about a view is that it's not a distinct container. Instead, it gives you a window on the current set of keys defined on the associated dictionary. If the dictionary changes, so does the view. For example:

```
>>> k = a.keys()
>>> k
dict_keys(['y', 'x', 'z'])
>>> a['t'] = 4
>>> k
dict_keys(['y', 'x', 'z', 't'])# Notice the change
>>>
```

Key-view objects also support a core group of set operations, including unions, intersections, and differences. This means that you can start to mix dictionary data with sets and easily perform more complex operations (e.g., identifying common keys, finding missing values, etc.). Here are some examples of such operations:

```
>>> b = { 'w' : 10,
        'x' : 20,
        'y' : 30 }
>>> a.keys() & b # Find keys in common
{'x', 'y'}
>>> a.keys() - b # Find keys in 'a', not in 'b'
{'t', 'z'}
>>> assert a.keys() == {'x','y','z','t'}
True
>>>
```

Such operations are actually highly relaxed in their type checking. In fact, they work if the other operand is any kind of sequence.

```
>>> a.keys() - ['x','y']
{'t', 'z'}
>>> a.keys() - "xy"  # "xy" is a sequence of chars 'x', 'y'
{'t', 'z'}
>>>
```

Tied into this whole picture are set and dictionary comprehensions. These mimic similar functionality found in list comprehensions. For example, here is a set comprehension:

```
>>> fruits = { 'pear', 'apple', 'banana', 'peach' }
>>> { f.upper() for f in fruits }
{'PEAR', 'BANANA', 'PEACH', 'APPLE'}
>>>
```

The above code runs about 30–40% faster than equivalent versions using list comprehensions such as `set([f.upper() for f in fruits])`. This is mainly due to not having to build the intermediate list first.

Dictionary comprehensions can be used similarly to construct new dictionaries. Here is an example that creates a dictionary consisting of keys not found in another dictionary:

```
>>> {key:a[key] for key in a.keys() - b }
{'z': 3, 't': 4}
>>>
```

I should note that general use of dictionary comprehensions seems to be a little tricky. Coming up with good use cases seems to be something that requires a bit more thought.

The New super()

For object-oriented programming, Python 3 features a new shorthand super() function that takes no arguments. For example:

```
class A:
      def bar(self):
         ...

class B(A):
      def bar(self):
         r = super().bar() # Call bar in parents
```

In past versions, you had to type `super(B,self).bar()`. Although this might seem like a minor feature, I find myself using it a lot—mainly taking advantage of the reduced typing.

Surprising Features

Certain subtle features have caught me by surprise. In most cases, I didn't discover these until I started porting libraries.

Changes to Exception Handling

Python 3 makes numerous changes to how exceptions get handled. Some changes are more subtle than others. For instance, the scope of exception variables has changed so that such variables do not exist beyond the associated `except` block:

```
>>> try:
      int("N/A")
    except ValueError as e:
      print("Error!")
```

```
Error!
>>> e  # Exists in Python 2, not in Python 3
Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
NameError: name 'e' is not defined
>>>
```

Similarly, exceptions can no longer be indexed as tuples:

```
>>> try:
      int("N/A")
    except ValueError as e:
      print(e[0]) # Works in Python 2, not in Python 3

Traceback (most recent call last):
      File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'N/A'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
      File "<stdin>", line 4, in <module>
TypeError: 'ValueError' object is not subscriptable
>>>
```

Both of these changes have the potential to break old code in a very subtle way—especially in unit tests involving exception handling.

The error message in the last example demonstrates yet another new feature of exception handling: chained exception messages. If an exception occurs while handling another exception, you will get a traceback that includes information about both exceptions (in this case, the original `ValueError` exception and the `TypeError` that occurred while trying to subscript the exception value). This is actually a welcome feature that should help people debug exceptions in complicated applications and libraries.

Bizarre Scoping Behavior of exec()

In Python 2, the exec() function executes a string of code as if it had been typed in place. For example, you could write the following code and it will produce exactly the output you expect:

```
def foo():
      y = 10
      exec("x = y + 42")
      print(x)  # Outputs 52 (in Python 2)
        # NameError exception in Python 3
```

In Python 3, exec() no longer executes in quite the same way—in fact, you'll get an exception if you try the above example. This is because it now executes the associated code in a dictionary that is a `copy` of the actual local variables (the result of the locals() function). If changes are made to any of the values in this dictionary, they are simply discarded instead of being written back to the original local variable. Thus, to execute the above code, you actually have to manage the variables yourself, as shown here:

```
def foo():
        y = 10
        lvars = locals()
        exec("x = y + 42",globals(),lvars)
        x = lvars['x'] # Get the changed value of x
        print(x)
```

Admittedly, use of exec() doesn't come up that often in most code, but if you do use it, you'll need to be aware that it doesn't work in quite the same way.

Subtle Differences Between Bytes and Strings

In Python 3, all strings are Unicode by default. However, there is a byte-string object for use with binary data. You might think that the Python 3 byte string is the same as the Python 2 string (str) object. This would be wrong. It actually behaves in some surprising ways, as shown below:

```
>>> s = b"Hello World"
>>> s[0] == b'H'
False
>>> s[0]
72
>>> s[:1]
b'H'
>>> t = bytes(13)
>>> t
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>>
```

In this example, you'll find that byte strings return integers when indexed but return new byte strings when sliced. In addition, converting values to bytes might not do what you expect (e.g., supplying an integer value to `bytes` makes a string of that size filled with zeroes).

Changes to bytes/Unicode handling have been, by far, the biggest headache in porting existing libraries to Python 3, especially libraries related to any kind of network programming or I/O. Not only do you have to make sure you're dealing with Unicode correctly, but you also need to account for the changed semantics of bytes to make sure your code works correctly.

The State of Third-Party Libraries

Perhaps the most major issue standing in the way of Python 3 adoption has been its lack of support for third-party libraries. A lot of Python programmers rely on packages such as numpy, Twisted, matplotlib, not to mention the plethora of Web frameworks. In this department, Python 3 is still a bit of a mixed bag and probably not for everyone.

Coming from the sciences, I have an interest in data analysis packages. For this, you're starting to see a lot more Python 3 support. For example, the popular numpy extension has supported Python 3 for at least the past year, and a recent coding sprint has been working to produce a Python 3–compatible version of matplotlib (https://github.com/matplotlib/matplotlib-py3).

For many modules, you can find experimental Python 3 support hidden away in a project fork or patch set. For example, if you're browsing around a project on a site such as GitHub, go look at the different project forks and pull requests. Sometimes you'll find an experimental Python 3 version just sitting there.

The one big caution with third-party packages is that much of the ported code is unproven or experimental. At this time, there just isn't a critical mass of Python 3 programmers to really iron out bugs. Thus you might find that you have to fix a lot of things on your own. For this, it helps to be comfortable with Python 3 itself (especially its I/O handling), makefiles, setup.py files, Python code, and even C programming. Frankly, it's probably best to keep your expectations low so that you aren't disappointed when something doesn't work as expected.

On the subject of low expectations, Web developers should probably avoid Python 3 for now. None of the large Web frameworks seems to support it or even provides a timeline of when Python 3 support might be added. If you're using a small HTTP framework or library, you might have more luck. For example, you can find Python 3 support in CherryPy (http://www.cherrypy.org/).

Final Words

Three years ago I recommended that it might be best to sit back and watch Python 3 development for a bit to see what happens. Today, my recommendation is not much different. If you're working with a lot of existing Python code involving various library dependencies, working in Python 3 certainly won't be easy. On the other hand, if you're starting something new or don't mind tinkering, you'll find a lot of neat stuff waiting for you. As for Python 3's future, the next few years should be interesting to watch as Python's grand experiment continues to unfold.

If you're interested in working in Python 3, there are a few books and resources of interest—for example, Lennart Regebro's *Porting to Python 3* (http://regebro.wordpress.com/porting-to-python-3) and the official "What's New in Python 3" documentation (http://docs.python.org/dev/whatsnew/). I have also given some PyCon tutorials on Python 3 I/O handling (http://www.dabeaz.com/python3io/) that may be useful for understanding some important issues that will arise in porting.

Finally, it's probably worth noting that much of the excitement in the Python world is currently focused on PyPy (http://pypy.org). PyPy is an implementation of Python 2.7 that features a just-in-time compiler and offers a substantial performance boost over CPython for many kinds of problems. Will PyPy be the future of Python? Only time will tell. However, I hope to take a closer look at PyPy in a future issue.