Some Easily Overlooked But Useful Python Features

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/index.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

For the past eight months, I've been locked away in my office working on a new edition of the *Python Cookbook* (O'Reilly & Associates). One of the benefits of writing a book is that you're forced to go look at a lot of stuff, including topics that you think you already know. For me, the *Cookbook* was certainly no exception. Even though I've been using Python for a long time, I encountered a lot of new tricks that I had never seen before. Some of these were obviously brand new things just released, but many were features I had just never noticed even though they've been available in Python for many years.

So, in this article, I'm going to take a tour through some of these easily overlooked features and show a few examples. Most of these features are extremely short—often one-liners that you can start using in your code. There's no particular order to the discussion; however, I do assume that you're using the latest version of Python, which is currently version 3.3. Many of the features presented will work in older versions, too.

Checking the Beginning and End of Strings

Sometimes you need to check the beginning or end of a string quickly to see whether it matches some substring. For example, maybe you've written some code that checks a URL like this:

```
# Check a URL for HTTP protocol
if url[:5] == 'http:' or url[:6] == 'https:':
    …

# Alternative using a regex
 if re.match('(http|https):', url):
     …
```

Sure, both solutions "work," but they're not nearly as simple as using the `startswith()` or `endswith()` method of a string. Just supply a tuple with all of the possible options you want to check. For example:

```
if url.startswith(('http:', 'https:')):
    …
```

Not only does this solution involve very little code, it runs fast and it's easy to read; however, you only get that benefit if you know that you can do it in the first place.

Tricks with format()

While I was teaching a training course a few years back, somebody pulled me aside to test me on their favorite job interview question for Python programmers. The problem was to write code that formatted an integer with the thousands comma separator properly placed in the right positions. I can only assume that he wanted me to write a solution like this:

```
>>> x = 1234567890
>>> print(','.join(reversed([str(x)[::-1][n:n+3][::-1]
            for n in range(0,len(str(x)),3)])))
1,234,567,890
>>>
```

Such problems are so much easier to solve if you just use `format()` like this:

```
>>> print(format(x, ','))
1,234,567,890
>>>
```

Ah, yes. That's much nicer. `format()` also works in ways that you might not expect with certain sorts of objects. For example, you can use it to format dates:

```
>>> from datetime import datetime
>>> d = datetime(2012, 12, 21)
>>> format(d, '%a, %b %d %m, %Y')
'Fri, Dec 21 12, 2012'
>>> format(d, '%a, %b %d, %Y')
'Fri, Dec 21, 2012'
>>> print('The apocalypse was on {:%Y-%m-%d}'.format(d))
The apocalypse was on 2012-12-21
>>>
```

Faster Date Parsing

On the subject of dates, I've recently learned that the common built-in function `strptime()` is dreadfully slow if you ever need to use it to parse a lot of dates. For example, suppose you were parsing a lot of date strings like this:

```
s = '16/Oct/2010:04:09:01'
```

The easiest way to parse it is to use `datetime.strptime()`. For example:

```
>>> import datetime
>>> d = datetime.strptime(s, '%d/%b/%Y:%H:%M:%S')
>>> d
datetime.datetime(2010, 10, 16, 4, 9, 1)
>>>
```

If you didn't know about such a function, you might be inclined to roll your own custom date parsing function from scratch. For example:

```
import calendar
months = {name:num for num, name in enumerate(calendar.
month_abbr)}

def parse_date(s):
      date, _, time = s.partition(':')
      day, mname, year = date.split('/')
      hour, minute, second = time.split(':')
      return datetime(int(year), months[mname], int(day),
                      int(hour), int(minute), int(second))
```

Here's an example of using the above function:

```
>>> d = parse_date(s)
>>> d
datetime.datetime(2010, 10, 16, 4, 9, 1)
>>>
```

More often than not, creating your own implementation of a function already built in to Python is a recipe for failure; however, not so in this case. It turns out that the custom `parse_date()` function runs nearly six times faster than `strptime()`. That kind of improvement can be significant in programs that are performing a lot of date parsing (e.g., parsing dates out of huge log files, data files, etc.).

One of the reasons `strptime()` is so slow is that it's actually written entirely in Python. Because it has to do a lot more work, such as interpreting the format codes, it's always going to be slower than a custom-crafted implementation aimed at a very specific date format.

New Time Functions

Not all is lost in the `time` module, however. Python recently picked up new timing-related functions. For making performance measurements, you can use the new `time.perf_counter()` function. For example:

```
import time
start = time.perf_counter()
...
end = time.perf_counter()
print('Took {} seconds'.format(end-start))
```

`perf_counter()` measures elapsed time using the most accurate timer available on the system. This eliminates some of the guesswork from benchmarking as common functions such as `time.time()` or `time.clock()`often have platform-related differences that affect their accuracy and resolution.

Similarly, the `time.process_time()` function can be used to measure elapsed CPU time. For example:

```
import time
start = time.process_time()
...
end = time.process_time()
print('Took {} CPU seconds'.format(end-start))
```

Last, but not least, the `time.monotonic()` function provides a monotonic timer where the reported values are guaranteed never to go backward—even if adjustments have been made to the system clock while the program is running.

All three of these time-related functions are only usable for working with time deltas. That is, you use them to compute time differences as shown. Otherwise, the value returned, although having a unit of seconds, doesn't have any useful meaning and may vary by platform.

Creating a File Only If It Doesn't Exist

Suppose you wanted to write to a file, but only if it doesn't exist. This is now easy in Python 3.3. Just give the 'x' file mode to `open()` like this:

```
>>> f = open('newfile.txt', 'x')
>>> f.write('Hello World')
>>> f.close()
>>>
>>> f = open('newfile.txt', 'x')
Traceback (most recent call last):
  File "", line 1, in
FileExistsError: [Errno 17] File exists: 'newfile.txt'
>>>
```

Although it's a simple feature, this saves you from first having to test like this:

```
import os.path
if not os.path.exists(filename):
    f = open(filename, 'w')
else:
    raise FileExistsError('File exists')
```

System Exit with Error Message

When writing scripts, it is common to follow a convention of writing a message to standard error and returning a non-zero exit code to report a failure. For example:

```
import sys

if must_die:
    sys.stderr.write('It failed!\n')
    raise SystemExit(1)
```

It turns out that all of the above code, including the import statement, can just be replaced by the following:

```
if must_die:
    raise SystemExit('It failed!')
```

This writes the message to standard error and exits with a code of 1. Who knew it was that easy? I didn't until recently.

Getting the Terminal Width

Sometimes you'd like to get the terminal width so that you can properly format text for output. To do this, you can try to fiddle around with environment variables, TTYs, and other details. Alternatively, you could just use the new `os.get_terminal_size()` function. For example:

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz.columns
108
>>> sz.lines
25
>>>
```

On the subject of formatting text for a terminal, the `textwrap` module can be useful. For example, suppose you had a long line of text like this:

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

You can use `textwrap.fill()` to reformat it:

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the
eyes,
not around the eyes, don't look around the eyes, look into my
eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.
```

Interpreting Byte Strings as Large Integers

Recently, I was working on a problem where I needed to parse and manipulate IPv6 network addresses such as "1234:67:89:aabb:43:210:dead:beef". I thought about writing some custom parsing code, but realized that it's probably better to do it using functions in the `socket` module:

```
>>> addr = "1234:67:89:aabb:43:210:dead:beef"
>>> import socket
>>> a = socket.inet_pton(socket.AF_INET6, addr)
>>> a
b'\x124\x00g\x00\x89\xaa\xbb\x00C\x02\x10\xde\xad\xbe\xef'
>>>
```

Yes, this "parsed" the IPv6 address, but it returned it as a 16-character byte-string representation of the 128-bit integer value. This is not quite what I had hoped for, so how was I going to turn such a string into a large integer value? It turns out it's trivial. Just use `int.from_bytes()` like this:

```
>>> int.from_bytes(a, 'big')
24196111521439464807328179944418033391
>>>
```

The second argument to `from_bytes()` is the byte order. Similarly, if you have a large integer value, you can go the other direction like this:

```
>>> x = 1234567890123456789012345678901234567890
>>> x.to_bytes(16, 'little')
b'\xd2\n?N\xee\xe0s\xc3\xf6\x0f\xe9\x8e\x01\x00\x00\x00'
>>> x.to_bytes(20, 'little')
b'\xd2\n?N\xee\xe0s\xc3\xf6\x0f\xe9\x8e\x01\x00\x00\x00\
x00\x00\x00\x00'
>>> x.to_bytes(20, 'big')
b'\x00\x00\x00\x00\x00\x00\x00\x01\x8e\xe9\x0f\xf6\xc3s\
xe0\xeeN?\n\xd2'
>>>
```

Manipulating Network Addresses

On the subject of manipulating network addresses, it became a whole lot easier in Python 3.3 with the addition of a new ipaddress library. Here's a short example of representing an IPv4 network and printing a list of all of the hosts contained within it:

```
>>> import ipaddress
>>> net = ipaddress.IPv4Network('192.168.2.0/29')
>>> net.netmask
IPv4Address('255.255.255.248')
>>> for n in net:
...     print(n)
...
192.168.2.0
192.168.2.1
192.168.2.2
192.168.2.3
192.168.2.4
192.168.2.5
192.168.2.6
192.168.2.7
>>> a = ipaddress.IPv4Address('192.168.2.14')
>>> a in net
False
>>> str(a)
'192.168.2.14'
>>> int(a)
3232236046
>>>
```

Calculating with Key Functions

At some point, most Python programmers encounter a problem where they need to sort some data. For example, suppose you had some stock data:

```
stocks = [ # (name, shares, price)
         ('AA', 100, 32.20),
         ('IBM', 50, 91.10),
         ('CAT', 150, 83.44),
         ('MSFT', 200, 51.23),
         ('GE', 95, 40.37),
         ('MSFT', 50, 65.10),
         ('IBM', 100, 70.44)
        ]
```

To sort the data, you can use the `sorted()` function; however, it only sorts according to the first tuple field (the name), producing this:

```
>>> sorted(stocks)
[('AA', 100, 32.2), ('CAT', 150, 83.44), ('GE', 95, 40.37), ('IBM', 50,
91.1),
 ('IBM', 100, 70.44), ('MSFT', 50, 65.1), ('MSFT', 200, 51.23)]
>>>
```

To change the sort, you can supply an optional "key" to `sorted()` like this:

```
>>> # sort by shares
>>> sorted(stocks, key=lambda s: s[1])
[('IBM', 50, 91.1), ('MSFT', 50, 65.1), ('GE', 95, 40.37), ('AA', 100,
32.2),
('IBM', 100, 70.44), ('CAT', 150, 83.44), ('MSFT', 200, 51.23)]

>>> # sort by price
>>> sorted(stocks, key=lambda s:s[2])
[('AA', 100, 32.2), ('GE', 95, 40.37), ('MSFT', 200, 51.23), ('MSFT',
50, 65.1),
('IBM', 100, 70.44), ('CAT', 150, 83.44), ('IBM', 50, 91.1)]
>>>
```

The key function is expected to take an element and return a value that's actually used to drive the sorting operation. In this example, the function is picking out the value of a specific column.

It's not as widely known, but the special key function can be given to a variety of other data-related functions. For example:

```
>>> # Find lowest price
>>> min(stocks, key=lambda s: s[2])
('AA', 100, 32.2)
```

```
>>> # Find maximum number of shares
>>> max(stocks, key=lambda s: s[1])
('MSFT', 200, 51.23)

>>> # Find 3 lowest prices
>>> import heapq
>>> heapq.nsmallest(3, stocks, key=lambda s:s[2])
[('AA', 100, 32.2), ('GE', 95, 40.37), ('MSFT', 200, 51.23)]
>>>
```

Final Words

That's about it for now. In the next issue, I'll plan to give a recap of highlights from the PyCon 2013 conference (held in March).