COLUMNS

Gleeful Incompatibility

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

Recently someone asked me when I thought that Python 2 and Python 3 might converge. They were a bit dismayed when I replied "never." If anything, Python 3 is moving farther and farther away from Python 2 at an accelerating pace. As I write this, Python 3.6 is just days from being released. It is filled with all sorts of interesting new features that you might want to use if you dare. Of course, you'll have to give up compatibility with all prior versions if you do. That said, maybe an upgrade is still worth it for your personal projects. In this article, I look at a few of the more interesting new additions. A full list of changes can be found in the "What's New in Python 3.6" document [1].

But First, Some Reflection

Since my earliest usage of Python, I've mostly viewed it as a personal productivity tool. I write a lot of custom scripts and use it for all sorts of tasks ranging from system administration to data processing. When I see new features, I think about how I might use them to make my life easier and more interesting. To be sure, this is a different view than that of a typical library writer who wants to maintain backwards compatibility with prior versions of Python. If you're mainly writing scripts for yourself, it is liberating to free yourself from the constraints of backwards compatibility. In this regard, Python 3.6 does not disappoint. However, if you're maintaining code for others, everything you're about to read should be taken with a grain of caution. So, with that said, let's begin!

String Formatting

Suppose you had a list of tuples like this

```
portfolio = [
  ('IBM', 50, 91.1),
  ('MSFT', 100, 63.45),
  ('HPE', 35, 42.75)
]
```

and you wanted to produce a nicely formatted table. There are many approaches to string formatting you might take. For example, you could use the classic string formatting operator (%):

```
>>> for name, shares, price in portfolio:
...     print('%10s %10d %10.2f' % (name, shares, price))
...
       IBM         50      91.10
      MSFT        100      63.45
       HPE         35      42.75
>>>
```

Or you could use the more verbose .format() method of strings:

```
>>> for name, shares, price in portfolio:
...     print('{:>10s} {:10d} {:10.2f}'.format(name, shares,
price))
...
       IBM        50      91.10
      MSFT       100      63.45
       HPE        35      42.75
>>>
```

Starting in Python 3.6, you can now use so-called "f-strings" to accomplish the same thing using far less code:

```
>>> for name, shares, price in portfolio:
...     print(f'{name:>10s} {shares:10d} {price:10.2f}')
...
       IBM        50      91.10
      MSFT       100      63.45
       HPE        35      42.75
>>>
```

f-strings are a special declaration of a string literal where expressions enclosed in braces are evaluated, converted to strings, and inserted into the resulting string [2]. In the above example, the name, shares, and price variables are picked up from the enclosing loop and inserted into the string. There's no need to use a special operator or method such as % or .format().

At first glance, it might appear that f-strings are a minor enhancement of what is already possible with the normal format() method. For example, format() already allows similar name substitutions:

```
>>> '{name:>10s} {shares:10d} {price:10.2f}'.
format(name=name, shares=shares, price=price)
'      HPE        35      42.75'
>>>
```

However, f-strings allow so much more. The greater power comes from the fact that nearly arbitrary expressions can be evaluated in the curly braces. For example, you can invoke methods and perform math calculations like this:

```
>>> f'{name.lower():>10s} {shares:10d} {price:10.2f}
{shares*price:10.2f}'
'      hpe        35      42.75    1496.25'
>>>
```

That's pretty neat and possibly rather surprising. For the most part, any expression can be placed inside the braces. The only restriction is that it cannot involve the backslash character (\). So attempts to mix f-strings and regular expressions might be thwarted. Of course, that's probably a good thing. Maybe.

Supervising Subclasses

Another interesting feature of Python 3.6 is the ability of a parent class to supervise the creation of child subclasses [3]. This can be done by providing a new special class method __init_subclass__(). For example, suppose you have this class:

```
class Base(object):
    @classmethod
    def __init_subclass__(cls):
        print('Base Child', cls)
        super().__init_subclass__()
```

Now, if you inherit from the class, you'll see the method fire:

```
>>> class A(Base):
...     pass
...
Base Child <class '__main__.A'>
>>> class B(A):
...     pass
...
Base Child <class '__main__.B'>
>>>
```

The use of super() in this example is to account for multiple inheritance. It allows for all of the parents to participate in the supervision if they want. For example, if you also had this class:

```
class Parent(object):
    @classmethod
    def __init_subclass__(cls):
        print('Parent Child', cls)
        super().__init_subclass__()
```

Now watch what happens with multiple inheritance:

```
>>> class C(Base, Parent):
...     pass
...
Base Child <class '__main__.C'>
Parent Child <class '__main__.C'>
>>>
```

Supervising subclasses might seem like a fairly esoteric feature, but it turns out to be rather useful in a lot of library and framework code because it can eliminate the need to use more advanced techniques such as class decorators or metaclasses. Here's an example that uses the __init_subclass__() method to register classes with a dictionary that's used in a convenience function.

```
class TableFormatter(object):
    _formats = {}
    @classmethod
    def __init_subclass__(cls):
        cls._formats[cls.name] = cls

def create_formatter(name):
    formatter_cls = TableFormatter._formats.get(name)
    if formatter_cls:
        return formatter_cls()
    else:
        raise RuntimeError('Bad format: %s' % name)

class TextTableFormatter(object):
    name = 'text'

class CSVTableFormatter(object):
    name = 'csv'

class HTMLTableFormatter(object):
    name = 'html'
```

In this code, the `TableFormatter` class maintains a registry of child classes. The `create_formatter()` function consults the registry and makes an instance using a short name. For example:

```
>>> create_formatter('csv')
<__main__.CSVTableFormatter object at 0x10ae9f748>
>>>
```

There are many other situations where a base class might want to supervise child classes. We'll see another example shortly.

Ordering Some (All?) of the Dicts

One of the more dangerously interesting features of Python 3.6 is that there are many situations where dictionaries are now ordered—preserving the order in which items were inserted. A dictionary like this

```
>>> s = { 'name': 'ACME', 'shares': 100, 'price': 385.23 }
>>>
```

now preserves the exact insertion order. This makes it much easier to turn a dictionary into a list or tuple in a way that respects the original structure of data. For example:

```
>>> keys = list(s)
>>> keys
['name', 'shares', 'price']
>>> row = tuple(s.values())
('ACME', 100, 385.23)
>>> dict(zip(keys, row))
{ 'name': 'ACME', 'shares': 100, 'price': 385.23 }
>>>
```

The fact that order is preserved may simplify a lot of data-handling problems: e.g., preserving the order of data found in files, JSON objects, and more. So, on the whole, it seems like a nice feature.

This ordering applies to other dictionary-related functionality. For example, if you write a function involving `**kwargs`, the order of the keyword arguments is preserved [4]:

```
>>> def func(**kwargs):
...     print(kwargs)
...
>>> func(spam=1, bar=2, grok=3)
{ 'spam': 1, 'bar': 2, 'grok': 3 }
>>>
```

Since the order is preserved, it seems to open up more possibilities for interesting functions involving `**kwargs`. For example, maybe you want to convert a sequence of lists to dictionaries:

```
rows = [
  ['IBM', '50', '91.1'],
  ['MSFT', '100', '63.45'],
  ['HPE', '35', '42.75']
]

def parse_rows(_rows, **columns):
  types = columns.values()
  names = columns.keys()
  for row in _rows:
     yield { name: func(val)
           for name, func, val in zip(names, types, row) }

for r in parse_rows(rows, name=str, shares=int, price=float):
    print(r)
```

Similarly, modules and classes now capture the definition order of their contents [5]. This is potentially useful for code that performs various forms of code introspection. For example, you can iterate over the contents of a class or module in definition order using a loop like this:

```
>>> import module
>>> for key, val in vars(module).items():
...     print(key, val)
...
>>>
```

As noted, this is one of the more dangerous features of Python 3.6. Past versions of Python do not guarantee dictionary ordering. So, if you rely upon this, know that your code will not work on any prior version. Also, the ordering seems to be provisional—meaning that it could be removed or refined in future Python versions.

Annotating All the Things

Since the earliest release of Python 3, it was possible for functions to have annotated arguments. For example:

```
def add(x:int, y:int) -> int:
    return x + y
```

The annotations didn't actually do anything, but served more as a kind of documentation. Tools could obtain the annotations by looking at the function's `__annotations__` attribute like this:

```
>>> add.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>, 'return': <class 'int'>}
>>>
```

The annotation idea is now extended to class attributes and variables [6]. For example, you can write a class like this:

```
class Point:
    x:int
    y:int
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Like their function counterparts, the annotations do nothing. They are merely collected in a class `__annotations__` `attribute`.

```
>>> Point.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>}
>>>
```

You can also annotate free-floating variables in a module. For example:

```
# spam.py

x:int = 0
y:int = 1
```

In this case, they become part of a module level `__annotations__` dictionary.

```
>>> import spam
>>> spam.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>}
>>>
```

It's important to note that the annotations don't change any aspect of Python's behavior. They are extra metadata that can be used by other tools such as frameworks, IDEs, or program checkers.

Summoning the Genie

Now that we've seen a few new features, it's time to gleefully put them into practice with something more interesting. How about a typed tuple object with a silly name?

```
import operator

class Toople(tuple):
    @classmethod
    def __init_subclass__(subcls):
        types = list(subcls.__annotations__.items())

        @staticmethod
        def __new__(cls, *args):
            if len(args) != len(types):
                raise TypeError(f'Expected {len(types)} args')
            for val, (name, ty) in zip(args, types):
                if not isinstance(val, ty):
                    raise TypeError(f'{name} must be an {ty.
__name__}')
            return super().__new__(cls, args)
        subcls.__new__ = __new__

        def __repr__(self):
            return f'{subcls.__name__}{super().__repr__()}'
        subcls.__repr__ = __repr__


        # Make properties for the attributes
        for n, name in enumerate(subcls.__annotations__):
            setattr(subcls, name, property(operator.itemgetter(n)))
```

Good god—f-strings, annotations, subclassing of the `tuple` built-in, and an `__init_subclass__` method that's patching child classes. What is going on here? Obviously, it's a small bit of Python 3.6 code that lets you write typed-tuple classes like this:

```
class Point(Toople):
    x:int
    y:int

class Stock(Toople):
    name:str
    shares:int
    price:float
```

Check it out:

```
>>> p = Point(2, 3)
>>> p
Point(2, 3)
>>> p.x
2
>>> p.y
3
```

```
>>> s = Stock('ACME', 50, 98.23)
>>> s
Stock('ACME', 50, 98.23)
>>> s.name
'ACME'
>>> s.shares
50
>>>

>>> Stock('ACME', '50', '98.23')
Traceback (most recent call last):
  ...
TypeError: shares must be an int
>>>
```

Okay, that's kind of awesome and insane. Don't try it on anything earlier than Python 3.6 though. It requires all of the features discussed including the reliance on newfound dictionary ordering. In fact, your coworkers might chase you out of the office while waving flaming staplers and hurling single-serve coffee packets at you if you put code like that in your current application. Nevertheless, it's a taste of what might be possible in the Python of the distant future.

Final Words

Over the last few years, a lot has been said about the Python 2 vs. Python 3 split. There are those who claim that Python 3 doesn't offer much that's new. Although that might have been true five years ago, it's becoming much less so now. In fact, Python 3 has all sorts of interesting new language features that you might want to take advantage of (e.g., I haven't even talked about the expanded features of async functions that were introduced in Python 3.5). Python 3.6 pushes all of this to a whole new level. Frankly, Python 3 has become a lot of fun that rewards curiosity and an adventurous spirit. If you're starting a new project, it's definitely worth a look.

References

[1] What's New in Python 3.6: https://docs.python.org/3.6/whatsnew/3.6.html.

[2] PEP 498—String literal interpolation: https://www.python.org/dev/peps/pep-0498/.

[3] PEP 487—Simpler customization of class creation: https://www.python.org/dev/peps/pep-0487/.

[4] PEP 468—Preserving the order of **kwargs in a function: https://www.python.org/dev/peps/pep-0468/.

[5] PEP 520—Preserving class attribute definition order: https://www.python.org/dev/peps/pep-0520/.

[6] PEP 526—Syntax for variable annotations: https://www.python.org/dev/peps/pep-0526/.