IBM

developerWorks.

Python 3 primer, Part 2: Advanced topics

Metaclasses, decorators, and other strange creatures

Skill Level: Intermediate

Cesar Otero
Freelance Consultant

30 Jan 2009

Python 3 is the latest version of Guido van Rossum's powerful general-purpose programming language. It breaks backwards compatibility with the 2.x line but has cleaned up some syntax issues. This second article builds on the previous article. In Part 2 of this two-part series, discover more new Python features and details on more advanced topics such as changes in abstract base classes, metaclasses, and decorators.

The previous article on Python version 3—also known as *Python 3000* or *Py3K*—discusses some of the basic changes in Python that will break backwards compatibility, such as the new `print()` function, the `bytes` data type, and changes to the `string` type. This article, Part 2, explores some of the more advanced topics, such as abstract base classes (ABCs), metaclasses, function annotations and decorators, integer literal support, the number type hierarchy, and changes to raising and catching exceptions, most of which also break backwards compatibility with the version 2x line.

Class decorators

In previous versions of Python, transformations to a method had to be done after the definition of the method. In longer methods, this requirement kept an important component of the definition far removed from the definition of the external interface provided in Python Enhancement Proposal (PEP) 318 (see Resources for a link). This snippet shows an example of this transformation requirement:

Listing 1. Method transformations in pre-version 3 Python

© Copyright IBM Corporation 2008. All rights reserved.

```
def myMethod(self):
    # do something

myMethod = transformMethod(myMethod)
```

To make situations like this more readable and to avoid having to reuse the same method name multiple times, method decorators were introduced in Python version 2.4.

Decorators are methods that modify other methods and return either a method or another callable object. They are denoted with an "at" symbol (@) before the name of the decorator—a similar syntax to Java™ annotations. Listing 2 shows decorators in action.

Listing 2. A decorator method

```
@transformMethod
def myMethod(self):
    # do something
```

Decorators are pure syntactic sugar—or (according to Wikipedia) "additions to the syntax of a computer language that do not affect its functionality but make it 'sweeter' for humans to use." A common use for decorators is to annotate static methods. Listing 1 and Listing 2 are equivalent, but Listing 2 is easier to read.

You define a decorator just like any other method:

```
def mod(method):
    method.__name__ = "John"
    return method

@mod
def modMe():
    pass

print(modMe.__name__)
```

Even better, Python 3 now supports decorators not only for methods, but also for classes. So, instead of using this:

```
class myClass:
    pass

myClass = doSomethingOrNotWithClass(myClass)
```

you can use this:

© Copyright IBM Corporation 2008. All rights reserved.

```
@doSomethingOrNotWithClass
class myClass:
    pass
```

Metaclasses

Metaclasses are classes whose instances are other classes. Python 3 has retained the built-in `metaclass` type, used to create other metaclasses or dynamically create classes at run time. It is still valid to use the syntax:

```
>>>aClass = type('className',
   (object,),
   {'magicMethod': lambda cls : print("blah blah")})
```

which accepts as its arguments a string as the class name, a tuple of inherited objects (which can be an empty tuple), and a dictionary (which can also can be empty) containing methods that you can add. Of course, you can also inherit from type and create your own metaclass:

```
class meta(type):
    def __new__(cls, className, baseClasses, dictOfMethods):
        return type.__new__(cls, className, baseClasses,
dictOfMethods)
```

Keyword-only arguments

Python 3 has changed how function arguments are assigned to "parameter slots." You can use an asterisk (`*`) in the passed-in parameters so that you don't accept variable-length arguments. Named arguments must follow the bare `*`—for example, `def meth(*, arg1): pass`. See the Python documentation or PEP 3102 for more information (see Resources for links).

Note: If the last two examples made no sense whatsoever, I highly recommend reading David Mertz and Michele Simionato's series on metaclasses. See Resources for a link.

Notice that now in a class definition, keyword arguments are allowed after the list of base classes—generally speaking, `class Foo(*bases, **kwds): pass`. The metaclass is passed in to the class definition, conveniently, using the keyword argument `metaclass`. For example:

```
>>>class aClass(baseClass1, baseClass2, metaclass = aMetaClass):
pass
```


© Copyright IBM Corporation 2008. All rights reserved.

The old syntax for metaclasses is to assign the metaclass to the built-in attribute `__metaclass__`:

```
class Test(object):
    __metaclass__ = type
```

Also, a new attribute—`__prepare__`—has been added. You use this attribute to create the dictionary for the new class namespace. It is called before the class body is evaluated, as Listing 3 shows.

Listing 3. A simple metaclass using the __prepare__ attribute

```
def meth():
    print("Calling method")

class MyMeta(type):
    @classmethod
    def __prepare__(cls, name, baseClasses):
        return {'meth':meth}

    def __new__(cls, name, baseClasses, classdict):
        return type.__new__(cls, name, baseClasses, classdict)

class Test(metaclass = MyMeta):
    def __init__(self):
        pass

    attr = 'an attribute'

t = Test()
print(t.attr)
```

A more interesting example, taken verbatim from PEP 3115 and shown in Listing 4, creates a metaclass with a list of names of its methods, while maintaining the order in which the class methods where declared.

Listing 4. A metaclass that preserves the order of the class members

```
# The custom dictionary
class member_table(dict):
    def __init__(self):
        self.member_names = []

    def __setitem__(self, key, value):
        # if the key is not already defined, add to the
        # list of keys.
        if key not in self:
            self.member_names.append(key)

        # Call superclass
        dict.__setitem__(self, key, value)

# The metaclass
class OrderedClass(type):
    # The prepare function
    @classmethod
```


© Copyright IBM Corporation 2008. All rights reserved.

```
    def __prepare__(metacls, name, bases): # No keywords in this
case
        return member_table()

    # The metaclass invocation
    def __new__(cls, name, bases, classdict):
        # Note that we replace the classdict with a regular
        # dict before passing it to the superclass, so that we
        # don't continue to record member names after the class
        # has been created.
        result = type.__new__(cls, name, bases, dict(classdict))
        result.member_names = classdict.member_names
        return result
```

There are a few reasons for the changes in metaclasses. Objects store their methods in a dictionary, which has no order. However, there are several cases in which preserving the order of declared class members would be useful. You do this by the metaclass getting "involved" earlier in the class creation, when the information is still available—useful, for example, in the creation of C structs. This new mechanism also allows for other interesting possibilities to be implemented in the future, such as inserting symbols into the body of the created class namespace during class construction and forward references of symbols. PEP 3115 also mentions that there are aesthetic reasons for changing the syntax, but this is a debate that cannot be solved objectively. (See Resources for a link to PEP 3115.)

Abstract base classes

As I mentioned in the previous article this series, ABCs are classes that can't be instantiated. Programmers coming from the Java or C++ languages should be familiar with this concept. Python 3 adds a new framework—abc—which provides support for working with ABCs.

The abc module has a metaclass (`ABCMeta`) and decorators (`@abstractmethod` and `@abstractproperty`). If an ABC has an `@abstractmethod` or an `@abstractproperty`, it cannot be instantiated but must be overridden in a subclass. For example, the code:

```
>>>from abc import *
>>>class C(metaclass = ABCMeta): pass
>>>c = C()
```

is okay, but don't do this:

```
>>>from abc import *
>>>class C(metaclass = ABCMeta):
...    @abstractmethod
...    def absMethod(self):
...        pass
>>>c = C()
Traceback (most recent call last):
```


© Copyright IBM Corporation 2008. All rights reserved.

```
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class C with abstract methods
absMethod
```

Even better, use the code:

```
>>>class B(C):
...     def absMethod(self):
...         print("Now a concrete method")
>>>b = B()
>>>b.absMethod()
Now a concrete method
```

The `ABCMeta` class overrides the attributes `__instancecheck__` and `__subclasscheck__`, providing a way of overloading the built-in functions `isinstance()` and `issubclass()`. To add a virtual subclass to your ABC, use the `register()` method that `ABCMeta` provides. The simple example:

```
>>>class TestABC(metaclass=ABCMeta): pass
>>>TestABC.register(list)
>>>TestABC.__instancecheck__([])
True
```

is equivalent to using `isinstance(list, TestABC)`. You might have noticed that Python 3 uses `__instancecheck__` instead of `__issubclass__`, and `__subclasscheck__` instead of `__issubclass__`, which seems more natural. The reasoning is that the reversal of the arguments `isinstance(subclass, superclass)`, for example, to `superclass.__isinstance__(subclass)` might cause confusion. So, the syntax `superclass.__instancecheck__(subclass)` wins.

Within the `collections` module, you can use several ABCs to test whether a class provides a specific interface:

```
>>>from collections import Iterable
>>>issubclass(list, Iterable)
True
```

Table 1 shows the ABCs of the collections framework.

Table 1. The ABCs of the collections framework

ABC	Inherits
Container	
Hashable	
Iterable	
Iterator	Iterable

© Copyright IBM Corporation 2008. All rights reserved.

`Sized`	
`Callable`	
`Sequence`	`Sized`, `Iterable`, `Container`
`MutableSequence`	`Sequence`
`Set`	`Sized`, `Iterable`, `Container`
`MutableSet`	`Set`
`Mapping`	`Sized`, `Iterable`, `Container`
`MutableMapping`	`Mapping`
`MappingView`	`Sized`
`KeysView`	`MappingView`, `Set`
`ItemsView`	`MappingView`, `Set`
`ValuesView`	`MappingView`

Collections

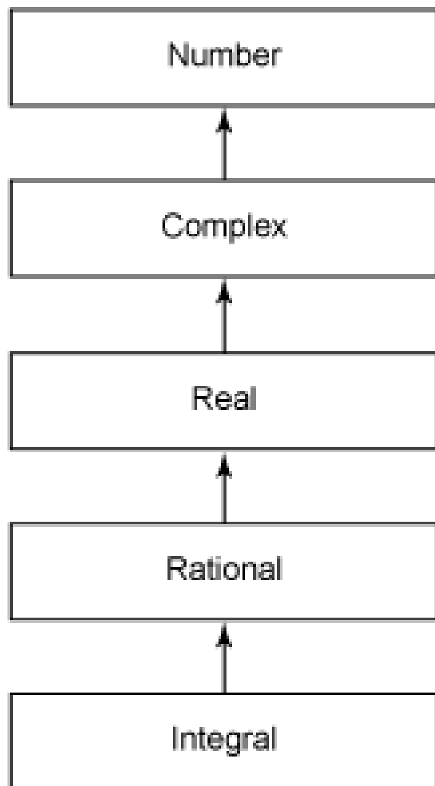
The collections framework consists of the container data types, double-ended queues (known as *deques*), and a default dictionary (known as a *`defaultdict`*). A deque supports appending and popping from either front or back. The `defaultdict` container, a subclass of the built-in dictionary (according to the Python 3 documentation) "overrides one method and adds one writable instance variable." Other than that, it acts like a dictionary. The collections framework also provides a data type factory function, `namedtuple()`.

The ABC type hierarchy

Python 3 now supports a type hierarchy of ABCs that represent numeric classes. These ABCs live in the `numbers` module and include `Number`, `Complex`, `Real`, `Rational`, and `Integral`. Figure 1 shows the number hierarchy. You can, of course, use these to implement your own numeric type or other numeric ABC.

Figure 1. The numeric hierarchy

© Copyright IBM Corporation 2008. All rights reserved.



The numerical tower
Python's numeric hierarchy was inspired by the Scheme language's numerical tower.

A new module, `fractions`, implements the numeric ABC `Rational`. This module gives support for rational number arithmetic. If you use `dir(fractions.Fraction)`, you'll notice it has attributes such as `imag`, `real`, and `__complex__`. Following the numeric tower, this is because `Rationals` inherit from `Reals`, which inherit from `Complex`.

Raising and catching exceptions

In Python 3, `except` clauses have been altered to deal with a syntactic ambiguity issue. Previously, in Python version 2.5, a `try . . . except` construction such as:

```
>>>try:
...     x = float('not a number')
... except (ValueError, NameError):
...     print "can't convert type"
```

might incorrectly be written as:

© Copyright IBM Corporation 2008. All rights reserved.

```
>>> try:
...     x = float('not a number')
... except ValueError, NameError:
...     print "can't convert type"
```

The problem with the latter form is that the `ValueError` exception will never be caught, because the interpreter will catch the `ValueError` and bind the exception object to the name `NameError`. This can be seen in the next example:

```
>>> try:
...     x = float('blah')
... except ValueError, NameError:
...     print "NameError is ", NameError
...
NameError is invalid literal for float(): not a number
```

So, to deal with the ambiguities, the comma (`,`) is substituted with the keyword `as` when you want to bind the exception object to another name. If you want to catch multiple exceptions, parentheses (`()`) are required. The code in Listing 5 shows two legitimate examples in Python 3.

Listing 5. Exception handling in Python 3

```
# bind ValueError object to local name ex
try:
    x = float('blah')
except ValueError as ex:
    print("value exception occurred ", ex)

# catch two different exceptions simultaneously
try:
    x = float('blah')
except (ValueError, NameError):
    print("caught both types of exceptions")
```

Another change in exception handling is *exception chaining*—either implicit or explicit. Listing 6 shows an example of an implicit exception chain.

Listing 6. An implicit exception chain in Python 3

```
def divide(a, b):
    try:
        print(a/b)
    except Exception as exc:
        def log(exc):
        fid = open('logfile.txt') # missing 'w'
        print(exc, file=fid)
        fid.close()

        log(exc)

divide(1,0)
```


© Copyright IBM Corporation 2008. All rights reserved.

The `divide()` method tries to perform a division by zero and raises an exception: `ZeroDivisionError`. But, in the exception clause, inside the nested `log()` method, `print(exc, file=fid)` attempts to write to a file that hasn't been opened for writing. Python 3 raises the exceptions shown in Listing 7.

Listing 7. The traceback of the implicit chained exception example

```
Traceback (most recent call last):
  File "chainExceptionExample1.py", line 3, in divide
  print(a/b)
ZeroDivisionError: int division or modulo by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "chainExceptionExample1.py", line 12, in <module>
    divide(1,0)
  File "chainExceptionExample1.py", line 10, in divide
    log(exc)
  File "chainExceptionExample1.py", line 7, in log
    print(exc, file=fid)
  File "/opt/python3.0/lib/python3.0/io.py", line 1492, in write
    self.buffer.write(b)
  File "/opt/python3.0/lib/python3.0/io.py", line 696, in write
    self._unsupported("write")
  File "/opt/python3.0/lib/python3.0/io.py", line 322, in
_unsupported
    (self.__class__.__name__, name))
io.UnsupportedOperation: BufferedReader.write() not supported
```

Notice that both exceptions are handled. In previous versions of Python, the `ZeroDivisionError` would have been lost. How is this accomplished? A `__context__` attribute such as `ZeroDivisionError` is now a part of all exception objects. In this case, the `__context__` attribute of the `IOError` that was raised "retains" the `ZeroDivisionError` in the `__context__` attribute.

In addition to the `__context__` attribute, exception objects have a `__cause__` attribute, which is always initialized to `None`. The purpose of this attribute is to give an explicit way to record the cause of an exception. The `__cause__` attribute is set with the following syntax:

```
>>> raise EXCEPTION from CAUSE
```

This is exactly the same as coding:

```
>>>exception = EXCEPTION
>>>exception.__cause__ = CAUSE
>>>raise exception
```

but much more elegant. The example in Listing 8 shows explicit exception chaining.

© Copyright IBM Corporation 2008. All rights reserved.

Listing 8. Explicit exception chaining in Python 3

```
class CustomError(Exception):
    pass

try:
    fid = open("aFile.txt") # missing 'w' again
    print("blah blah blah", file=fid)
except IOError as exc:
    raise CustomError('something went wrong') from exc
```

As in the previous example, the `print()` function raises an exception, because the file was not open for writing. Listing 9 shows the traceback.

Listing 9. The exception traceback

```
Traceback (most recent call last):
  File "chainExceptionExample2.py", line 5, in <module>
    fid = open("aFile.txt")
  File "/opt/python3.0/lib/python3.0/io.py", line 278, in __new__
    return open(*args, **kwargs)
  File "/opt/python3.0/lib/python3.0/io.py", line 222, in open
    closefd)
 File "/opt/python3.0/lib/python3.0/io.py", line 615, in __init__
    _fileio._FileIO.__init__(self, name, mode, closefd)
IOError: [Errno 2] No such file or directory: 'aFile.txt'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
File "chainExceptionExample2.py", line 8, in <modulei>
  raise CustomError('something went wrong') from exc
__main__.CustomError: something went wrong
```

Notice in the traceback the line, "The above exception was the direct cause of the following exception," which is followed by another traceback that results in the `CustomError`, "something went wrong."

Finally, yet another attribute added to exception objects is the `__traceback__` attribute. If a caught exception doesn't have its `__traceback__` attribute, then the new traceback is set. Here's a simple example:

```
from traceback import format_tb

try:
    1/0
except Exception as exc:
    print(format_tb(exc.__traceback__)[0])
```

Notice that `format_tb` returns a list and that there is only one exception in this list.

The with statement

As of Python version 2.6, `with` is now a keyword, and this feature

© Copyright IBM Corporation 2008. All rights reserved.

no longer needs to be imported through `__future__`.

Integer literal support and syntax

Python supports string literals of integers of different bases—octal, decimal (obviously!), and hexadecimal—and now binary has been added. The representation for octal literals has changed: Octal literals are now represented with either a prefixed `0o` or `0O` (that is, a zero followed by either an uppercase or lowercase *o*) and the literal to be evaluated. For example, an octal 13 or decimal 11 is represented as:

```
>>>0o13
11
```

The new binary literals are represented with either a prefixed `0b` or `0B` (that is, a zero followed by an uppercase or lowercase *b*). The representation of decimal 21 in binary is then:

```
>>>0b010101
21
```

The `oct()` and `hex()` methods have been removed.

Function annotations

Function annotations associate expressions with parts of a function, such as parameters, at compile time. Left to themselves, function annotations are meaningless—that is, they aren't processed unless a third-party library does so. The purpose of function annotations is to standardize the way a function's parameters or return values are annotated. The syntax for function annotations is:

```
def methodName(param1: expression1, ..., paramN:
expressionN)->ExpressionForReturnType:
    ...
```

For example, here are annotations for a function's parameters:

```
def execute(program:"name of program to be executed", error:"if
something goes wrong"):
    ...
```


© Copyright IBM Corporation 2008. All rights reserved.

The following example annotates the return value of a function. This is useful for checking the return type of a function:

```
def getName() -> "isString":
    ...
```

The full grammar for function annotations can be found in PEP 3107 (see Resources for a link).

Conclusion

An Easter egg
In my opinion, the best module added to Python 3 is `antigravity`. Start Python, then type `import antigravity` at the command line. You won't be disappointed.

The final release for Python 3 came out in early December 2008. Since then, I've had a chance to check over some of the blogs and how people have been reacting to the backwards-incompatibility issue. Although I can't claim to have an official consensus by any measure, the blogs I've read certainly seem polarized. Some in the Linux® development community really don't seem to like the transition to version 3 because of the massive amount of code that has to be ported. In contrast, many Web developers will make the transition because of the changes in unicode support alone.

I will argue, at the very least, that before you pass judgement, you should look through the PEPs and the development mailing lists before deciding not to port to the new version. The PEPs explain the rationales for a particular change and the benefits reaped as well as the implementation. These changes were really well thought out and discussed to death. The topics covered in this series have been presented so that the common Python programmer can get a quick feel for the changes without going through all the PEPs.

© Copyright IBM Corporation 2008. All rights reserved.

Resources

Learn

- Start with Part 1 in this two-part series: Python 3 primer, Part 1: What's new.
- Read up on the relevant Python 3 PEPs:
  - PEP 318: Decorators for Functions and Methods
  - PEP 3107: Function Annotations
  - PEP 3129: Class Decorators
  - PEP 3127: Integer Literal Support and Syntax
  - PEP 3115: Metaclasses in Python 3000
  - PEP 3119: Introducing Abstract Base Classes
  - PEP 3141: A Type Hierarchy for Numbers
  - PEP 3109: Raising Exceptions in Python 3000
  - PEP 3110: Catching Exceptions in Python 3000
  - PEP 3102: Keyword-Only Arguments
- Read Wikipedia's entry on metaclasses.
- Read David Mertz and Michele Simionato's developerWorks series, Metaclass programming in Python (developerWorks, February 2003).
- Check out the Python 3 help files.
- Wikipedia provides a nice explanation of deques.

Get products and technologies

- Get the latest version of Python.

About the author

Cesar Otero
Cesar Otero is a freelance Java and Python consultant. He holds a degree in electrical engineering with a minor in mathematics.

© Copyright IBM Corporation 2008. All rights reserved.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of IBM trademarks.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

© Copyright IBM Corporation 2008. All rights reserved.