Python 3 primer, Part 1: What's new

Cleaner syntax for better code

Skill Level: Intermediate

Cesar Otero
Freelance Consultant

19 Dec 2008

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 article is the first in a series that talks about the changes that affect the language and backwards compatibility, and it provides examples of new features.

Python version 3, also known as *Python 3000* or *Py3K* (a nickname that's a pun on the Microsoft® Windows® 2000 operating system), is the latest version of Guido van Rossum's general-purpose programming language. Although many improvements have been made to the core language, the new version will break backwards compatibility with the 2.x line. Other changes have been anticipated for a while, such as:

- True division—for example, *1/2* returns *.5*.
- The `long` and `int` types have been unified into a single type, with the trailing *L* now removed.
- `True`, `False`, and `None` are now keywords.

This article—the first in a series on Python 3—covers the new `print()` function, `input()`, changes to input/output (I/O), the new `bytes` data type, changes to strings and string formatting, and finally, changes to the built-in `dict` type. This article is meant for programmers already familiar with Python who are curious about the changes but don't want to wade through the long list of Python Enhancement Proposals (PEPs).

The new print() function

You'll have to retrain your fingers to stop typing `print "hello"` and start typing `print("hello")`, because `print` is now a function, not a statement. I know, it's painful. Every Python programmer I know—as soon as they install version 3 and get the error "incorrect syntax"—screams in agony. I know the extra two characters are annoying; I know this will break backwards compatibility. But there are advantages.

Consider cases in which you need to redirect standard output (stdout) to a log. The following example opens the file log.txt for appending and assigns the object to `fid`. A string is then redirected to the file, `fid`, using `print>>`:

```
>>>fid = open("log.txt", "a")
>>>print>>fid, "log text"
```

Another example is to redirect to standard error (sys.stderr):

```
>>>print>>sys.stderr, "an error occurred"
```

Both of the previous examples are nice, but there's a better solution. The new syntax is now to simply pass in a value to the keyword argument `file` in the `print()` function. For example:

```
>>>fid = open("log.txt", "a")
>>>print("log.txt", file=fid)
```

This code has much cleaner syntax. Another advantage is the ability to change the separator by passing a string to the `sep` keyword argument and change the end string by passing another string to the `end` keyword argument. To change the separator, you could use:

```
>>>print("Foo", "Bar", sep="%")
>>>Foo%Bar
```

In general, the new syntax is:

```
print([object, ...][, sep=' '][,
end='endline_character_here'][, file=redirect_to_here])
```

where the code inside the square brackets (`[]`) is optional. By default, calling `print()` by itself appends a newline character (`\n`).

From raw_input() to input()

In Python version 2.x, `raw_input()` reads an input from standard input (sys.stdin) and returns a string with the trailing newline character stripped from the end. The following example uses `raw_input()` to grab a string from the command prompt, then assigns the value to `quest`.

```
>>>quest = raw_input("What is your quest? ")
What is your quest? To seek the holy grail.
>>>quest
'To seek the holy grail.'
```

In contrast, the `input()` function in Python 2.x expects a valid Python expression, such as *3+5*.

Originally, it was suggested that both `input()` and `raw_input()` be removed from the Python built-in namespace altogether, thereby requiring an import to have any kind of input capability. This seemed pedagogically unsound; suddenly, simply typing:

```
>>>quest = input("What is your quest?")
```

would have turned into:

```
>>>import sys
>>>print("What is your quest?")
>>>quest = sys.stdin.readline()
```

which is far more verbose for just a simple input and a lot more to explain to a novice. This would have required teaching what *modules* and *imports* are, printing a string, and the dot operator. (This suspiciously feels a little too much like the Java™ language...) So, in Python 3, `raw_input()` is renamed `input()`, and no import is required to get data from the standard input. If you need to retain the version 2.x `input()` functionality, use `eval(input())`, which works identically.

A bit on bytes

The new data type, the bytes literal, as well as the `bytes` object are used for storing binary data. This object is an immutable sequence of integers between 0 and 127, or ASCII-only characters. Really, it's an immutable version of the `bytearray` object from version 2.5. A *bytes literal* is a string with a *b* before it—for example, *b'byte literal'.* Evaluation of a bytes literal produces a new `bytes` object. You can create a new `bytes` object with the `bytes()` function. The constructor for a `bytes` object is:

```
bytes([initializer[, encoding]])
```

For example:

```
>>>b = (b'\xc3\x9f\x65\x74\x61')
>>>print(b)
b'\xc3\x83\xc2\x9feta'
```

creates a `bytes` object but is redundant, because you can create a `bytes` object by simply assigning a byte literal. (I just wanted to demonstrate that you can do this: I'm not actually suggesting you do it.) If you wanted to use an iso-8859-1 encoding, you could try this:

```
>>>b = bytes('\xc3\x9f\x65\x74\x61', 'iso-8859-1')
>>>print(b)
b'\xc3\x83\xc2\x9feta'
```

If the initializer is a string, you must provide an encoding. If the initializer is a bytes literal, you need not specify the encoding type: Remember, bytes literals are not strings. But like strings, you can concatenate bytes:

```
>>>b'hello' b' world'
b'hello world'
```

You use the `bytes()` method to represent both binary data and encoded text. To convert from `bytes` to `str`, the `bytes` object must be decoded (more on this later). Binary data is decoded with the `decode()` method. For example:

```
>>>b'\xc3\x9f\x65\x74\x61'.decode()
'ßeta'
```

You can also read binary data directly from a file. The code:

```
>>>data = open('dat.txt', 'rb').read()
>>>print(data) # data is a string
>>># content of data.txt printed out here
```

opens for reading a file object in binary mode and reads in the entire file.

Strings

Python has a single string type, `str`, that behaves similarly to the version 2.x

unicode type. In other words, all strings are unicode strings. Also—and very conveniently for non-Latin text users—non-ASCII identifiers are now permissible. For example:

```
>>>césar = ["author", "consultant"]
>>>print(césar)
['author', 'consultant']
```

In previous versions of Python, the `repr()` method converts 8-bit strings to ASCII. For example:

```
>>>repr('é')
"'\\xc3\\xa9'"
```

It now returns a unicode string:

```
>>>repr('é')
"'é'"
```

which, as mentioned earlier, is the built-in string type.

String objects and byte objects are incompatible. If you want the string representation of a byte, use its `decode()` method. Conversely, use the `encode()` method of a string object if you want a bytes literal from that string.

Changes in string formatting

Many Python programmers felt that the built-in % operator for formatting strings was too constrained, because:

- It is a binary operator and can take at most two arguments.
- Exempting the format string argument, all other arguments must be squeezed in with either a tuple or a dictionary.

This style is somewhat inflexible, so Python 3 introduces a new way of doing string formatting. (Both the % operator and the `string.Template` module are retained in version 3.) String objects now have a method, `format()`, that accepts positional and keyword arguments, which are passed into *replacement fields.* Replacement fields are denoted by curly brackets (`{}`) inside a string. The element inside a replacement field is simply called a *field.* Here's a simple example:

```
>>>"I love {0}, {1}, and {2}".format("eggs", "bacon",
```

```
"sausage")
'I love eggs, bacon, and sausage'
```

The fields **{0}**, **{1}**, and **{2}** are passed in the positional parameters `eggs`, `bacon`, and `sausage` to the `format()` method. The following example shows how to use `format()` with keyword arguments passed in to format:

```
>>>"I love {a}, {b}, and {c}".format(a="eggs", b="bacon",
c="sausage")
'I love eggs, bacon, and sausage'
```

Here's another example that combines positional parameters and keyword arguments:

```
>>>"I love {0}, {1}, and {param}".format("eggs", "bacon",
param="sausage")
'I love eggs, bacon, and sausage'
```

Remember that it's a syntax error to have a non-keyword argument after a keyword argument. To escape the curly braces, double them, like so:

```
>>>"{{0}}".format("can't see me")
'{0}'
```

The positional parameter `can't see me` isn't printed, because there's no field to print to. Note that this does not cause an error.

The new `format()` built-in function formats a single value. For example:

```
>>>print(format(10.0, "7.3g"))
       10
```

In other words, the *g* stands for *general format*, which prints a fixed-width number. The first number before the dot specifies the minimum width, and the number after the dot specifies the precision. The complete syntax for format specifiers is beyond the scope of this article, but you can find links to more information in the Resources section.

Changes to the built-in dict type

Another major change in 3.0 is the removal of the `dict.iterkeys()`, `dict.itervalues()`, and `dict.iteritems()` methods in dictionaries. Instead, you use `.keys()`, `.values()`, and `.items()`, which have been revamped to

return lightweight, set-like container objects instead of a list that's a copy of the keys or values. The advantage here is the ability to perform `set` operations on keys and items without having to copy them. For example:

```
>>>d = {1:"dead", 2:"parrot"}
>>>print(d.items())
<built-in method items of dict object at 0xb7c2468c>
```

Note: In Python, *sets* are unordered collections of unique elements.

Here, I created a dictionary with two keys and values, then printed the values of `d.items()`, which returns an object, not a list of values. You can test the membership of an element just like a `set` object:

```
>>>1 in d # test for membership
True
```

Here's an example of iterating over the items of the `dict_values` object:

```
>>>for values in d.items():
...     print(values)
...
dead
parrot
```

But, if you really want a list of values, you can always cast the returned `dict` object. For example:

```
>>>keys = list(d.keys())
>>>print(keys)
[1,2]
```

New I/O

> **Metaclasses**
> According to Wikipedia, "a *metaclass* is a class whose instances are classes." I explore this concept in more detail in part 2 of this series.

Before delving in to the new mechanisms for I/O, it's necessary to review abstract base classes (ABCs). A more in-depth treatment is provided on this topic in the second part of this series.

ABCs are classes that can't be instantiated. To use an ABC, a subclass must inherit

from the ABC and override its abstract methods. A method is abstract if it's preceded with the decorator `@abstractmethod`. The new ABC framework also provides the `@abstractproperty` decorator for defining abstract properties. You access the new framework by importing the standard library module `abc`. Listing 1 provides a simple example.

Listing 1. A simple abstract base class

```
from abc import ABCMeta

class SimpleAbstractClass(metaclass=ABCMeta):
    pass

SimpleAbstractClass.register(list)

assert isinstance([], SimpleAbstractClass)
```

The `register()` method call takes a class as an argument and makes the ABC a subclass of the registered class. You can verify this by calling the `assert` statement on the last line. Listing 2 provides another example that uses decorators.

Listing 2. An implemented abstract base class with decorators

```
from abc import ABCMeta, abstractmethod

class abstract(metaclass=ABCMeta):
    @abstractmethod
    def absMeth(self):
        pass

class A(abstract):
    # must implement abstract method
    def absMeth(self):
        return 0
```

Now that you know about ABCs, let's continue with the new I/O system. Previous Python releases lacked important yet exotic functions, such as `seek()`, for some stream-like objects. *Stream-like objects* are file-like objects with `read()` and `write()` methods—sockets or files, for example. Python 3 has multiple layers for I/O on stream-like objects—a raw I/O layer, a buffered I/O layer, and a text I/O layer—each defined with it own ABC with implementations.

You still open a stream using the built-in `open(fileName)` function, although you can also call `io.open(fileName))`. Doing so returns a buffered text file; `read()` and `readline()` return strings. (Remember that all strings in Python 3 are unicode.) You can also open a buffered binary file by using the form `open(fileName, 'b')`. In this case, `read()` returns bytes, but you can't use `readline()`.

The constructor for the built-in `open()` function is:

```
open(file,mode="r",buffering=None,encoding=None,errors=None,newline=None,closefd=True)
```

The possible modes are:

- **r:** Reading
- **w:** Open for writing
- **a:** Open for appending
- **b:** Binary mode
- **t:** Text mode
- **+:** Open a disk file for updating
- **U:** Universal newline mode

The default mode is `rt`, or open for reading text mode.

The `buffering` keyword argument expects one of three integers to determine the buffering policy:

- **0:** Switches buffering off
- **1:** Line buffering
- **> 1:** Full buffering (default)

The default encoding is platform dependent. The close file descriptor, or `closefd`, can be True or False. If False, the file descriptor is kept after the file is closed. Providing a file name won't work, in which case, `closefd` must be set to True.

The object that `open()` returns depends on the mode you set. Table 1 shows the return types.

Table 1. Return types for different open modes

Mode	Object returned
Text mode	`TextIOWrapper`
Binary	`BufferedReader`
Write binary	`BufferedWriter`
Append binary	`BufferedWriter`
Read/Write mode	`BufferedRandom`

Note: Text mode can be `w`, `r`, `wt`, `rt`, and so on.

The example in Listing 3 opens a buffered binary stream for reading.

Listing 3. Open a buffered binary stream for reading

```
>>>import io
>>>f = io.open("hashlib.pyo", "rb")  # open for reading in
binary mode
>>>f                                 # f is a
BufferedReader object
<io.BufferedReader object at 0xb7c2534c>
>>>f.close()                        # close stream
```

The `BufferedReader` object has access to several useful methods, such as `isatty`, `peek`, `raw`, `readinto`, `readline`, `readlines`, `seek`, `seekable`, `tell`, `writable`, `write`, and `writelines`, to name a few. To see the full list, run a `dir()` on a `BufferedReader` object.

Conclusion

Whether the Python community will accept version 3 is anyone's guess. The breaking of backwards compatibility will mean supporting two different versions in parallel. Some project developers may not want to migrate their projects, even with the 2to3 converter. Personally, I found that migrating from Python version 2 to 3 was primarily a matter of relearning a few things: It certainly wasn't as drastic a change as moving from Python to say the Java or Perl languages. Many of the changes have been long anticipated, such as true division and changes to `dict`. Performing a `print()` is a whole lot easier than `System.out.println()` in Java, so the learning curve is relatively small and there are advantages to be gained.

I'm guessing from reading entries in the blogosphere that many Pythonistas consider some of the changes—such as the backwards compatibility break—deal breakers. Lambda had originally been scheduled for removal but has been retained, and in its original form. For the complete list of things that are staying, visit the Python core development site. If you're adventurous enough to rummage through the PEPs, you can find more in-depth information there.

The next installment in this series will cover more advanced topics, such as metaclass syntax, ABCs, decorators, integer literal support, base types, and exceptions.

Resources

Learn

- See the Python core development site for the complete syntax for format specifiers.
- Read up on the relevant Python 3 PEPs:
  - PEP 3111: Simple input built-in in Python 3000
  - PEP 3116: New I/O
  - PEP 3138: String representation in Python 3000
  - PEP 3112: Bytes literals in Python 3000
  - PEP 3137: Immutable Bytes and Mutable Buffer
  - PEP 3106: Revamping dict.keys(), .values() & .items()
  - PEP 3108: Standard Library Reorganization
  - PEP 3100: Miscellaneous Python 3.0 Plans
- See metaclasses on Wikipedia.
- Review computer classes, including the abstract classes, on Wikipedia.
- Read Guido van Rossum's essays.
- Read up on Python's unordered collections of unique elements, or *sets.*
- In the developerWorks Linux zone, find more resources for Linux developers (including developers who are new to Linux), and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Get the latest version of Python.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

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.

Trademarks