What's new in Python 3

THE NEXT SNAKE

What do Python 2.x programmers need to know about Python 3? **BY RAINER GRIMM**

With the latest major Python release, creator Guido van Rossum saw the opportunity to tidy up his famous scripting language. What is different about Python 3.0? In this article, I offer some highlights for Python programmers who are thinking about making the switch to 3.x.

The first important point is that version 3.0, which is also known as Python 3000 or Py3k, broke with an old tradition in that it is not downwardly compatible. According to von Rossum, "There is no requirement that Python 2.6 code will run unmodified on Python 3.0."

Python 2.6's main purpose is to take the pain out of switching versions. Many of Python 3.0's features were backported to Python 2.6. (A similar Python 2.7 version will accompany the recent Python 3.1 release.) In addition to these transitional 2.x versions, the *2to3* command-line tool supports programmers migrating code from Python 2.x to Python 3.x.

Quirks

Python has some quirks that have bugged van Rossum since the beginning of the millennium. With Python 3, he decided to sacrifice downward compatibility to remove these obstacles. The most blatant break with Python 2 is the syntactic change to *print*, which has mutated from a statement to a function; thus, parameters, now called keywords, are called in parentheses. This change is in line with Tim Peters' *The Zen of Python*: "Explicit is better than implicit." [1]. The new generic print syntax is *print(*args, sep = ' ', end = '\n', file = sys.stdout)*, where *args* are the arguments, *sep* is the separator between the arguments, *end* is the end-of-line character, and *file* is the output medium.

Table 1: Print Syntax

Examples	Python 2.*	Python 3.*
Generic form	print "x = ", 5	print ("x = ", 5)
Create newline	print	print()
Suppress newline	print x,	print(x, end=" ")
Suppress newline without blanks		print(1,2,3,4,5, sep="")
Redirect output	print >>sys.stderr, "fatal error"	print("fatal error", file=sys.stderr)

Table 1 lists the syntax changes to the *print* function, including their default values. The advantage of the new version is not apparent until you take a closer look; the new print function now supports overloading. Listing 1 shows a print function that writes to standard output and to a logfile at the same time. To allow this to happen, the function instrumentalizes the built-in *__builtins__.print* function.

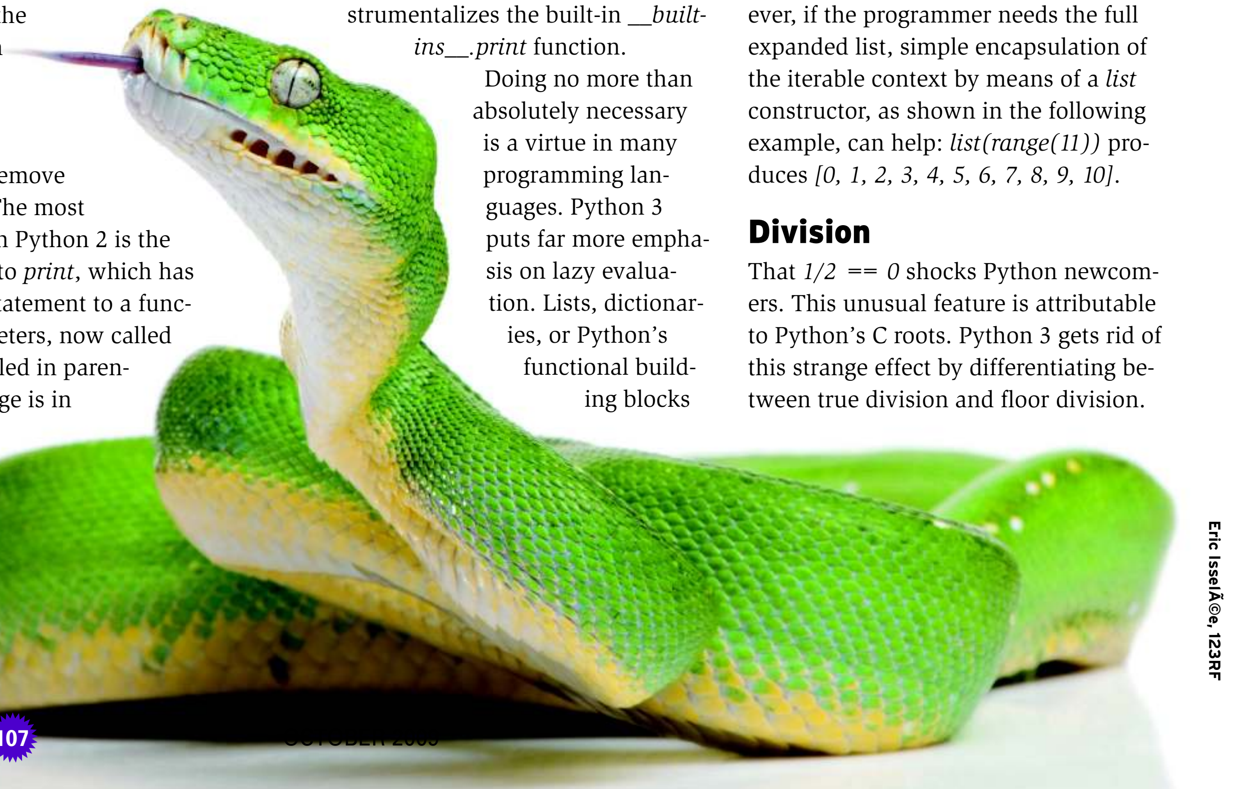
Doing no more than absolutely necessary is a virtue in many programming languages. Python 3 puts far more emphasis on lazy evaluation. Lists, dictionaries, or Python's functional building blocks no longer generate complete lists; they add just enough to support the evaluation of the output. Lazy evaluation thus saves memory and time.

The Python interpreter achieves this effect by returning only one iterable context that generates values on request. (This was what distinguished *range* and *xrange* in Python 2.) In Python 3, *range* acts like *xrange*, thus making the *xrange* function redundant.

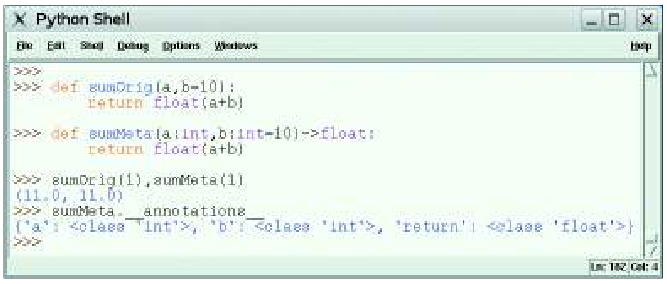
The same applies to the functional building blocks *map*, *filter*, and *zip*. These functions have been replaced by their equivalents from the *itertools* library. For dictionaries, the resulting, iterable contexts are known as views. However, if the programmer needs the full expanded list, simple encapsulation of the iterable context by means of a *list* constructor, as shown in the following example, can help: *list(range(11))* produces *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*.

Division

That *1/2 == 0* shocks Python newcomers. This unusual feature is attributable to Python's C roots. Python 3 gets rid of this strange effect by differentiating between true division and floor division.



Eric IsselÃ©e, 123RF

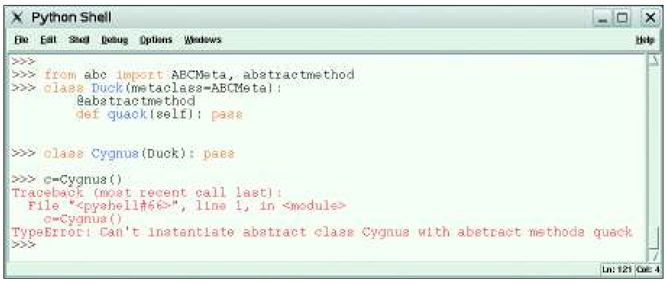


```
>>>
>>> def sumOrig(a,b=10):
        return float(a+b)

>>> def sumMeta(a:int,b:int=10)->float:
        return float(a+b)

>>> sumOrig(1),sumMeta(1)
(11.0, 11.0)
>>> sumMeta.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'float'>}
>>>
```

Figure 1: Function annotations bind metadata to a function.



```
>>>
>>> from abc import ABCMeta, abstractmethod
>>> class Duck(metaclass=ABCMeta):
        @abstractmethod
        def quack(self): pass


>>> class Cygnus(Duck): pass

>>> c=Cygnus()
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    c=Cygnus()
TypeError: Can't instantiate abstract class Cygnus with abstract methods quack
>>>
```

Figure 2: Failure to instantiate.

Whereas in true division *1/2 == 0.5*, floor division works like division in Python 2. The notation for floor division uses two slashes: *1//2 == 0*.

In Python 2, programmers had to declare strings explicitly as Unicode strings. Python 3 strings are now implicitly Unicode. Python 3.0 only distinguishes between text and data. Text (*str*) relates to strings and is the same as Python 2 Unicode strings. Data (*bytes*) are 8-bit strings and the same as Python 2 strings. Python 3 developers need to declare data: *b"8_bit_string"*. Table 2 gives an overview of the difference between Python 2 and 3.

The *str.encode()* and *bytes.decode()* functions are available to convert between data types. Programmers need this conversion in Python 3.0 when working with both data types; implicit type conversion no longer takes place. Programmers simply have to decide.

Syntactic Changes

Function annotations in Python 3 now allow programmers to bind metadata to a function. Decorators can then be added to the function in a second step; decorators automatically generate data from the metadata or perform type checking at run time.

The equivalent functions, *sumOrig* and *sumMeta* (Figure 1), show function declarations with and without metadata. The second function includes metadata for the signature and return value. This metadata can be referenced with the *__annotations__* function attribute.

Backports

Python 2.6's whole purpose in life is to make the move to version 3 as easy as possible, so many Python 3.0 features were backported to Python 2.6.

Resource management using *with* is an important new feature in Python 2.6. A resource (file, socket, mutex, …) is automatically bound when entering and released when exiting the *with* block. This idiom will remind C++ programmers of "Resource Acquisition Is Initialization" (RAII), wherein a resource is bound to an object. The *with* statement acts like *try … finally* from the user's point of view, in that both the *try* block and the *finally* block are always executed. All of this happens without explicit exception handling.

How does all of this work? In a *with* block, you can use any object provided by the context management protocol – that is, which has internal *__enter()__* and *__exit()__* methods. When entering the *with* block, the *__enter()__* method is automatically called, as is *__exit()__* when exiting the block.

The file object comes complete with these methods (Listing 2). Resource management is easy to code, however. A classic application would be to protect a code block against simultaneous access. Listing 3 has placeholders for the code block. The *locked* class objects prevent competitive access in the *with* block by using *myLock* to synchronize the block.

If this is too much work for you, you can use the *contextmanager* decorator from the new *contextlib* [2] library to benefit from its resource management functionality. Many other applications are listed in the Python Enhancement Proposal (PEP) 0343 [3].

Abstractions

The biggest syntactic extension to Python 2.6 is probably the introduction of abstract base classes. Whether an object could be used in a context previously depended on the object's properties and not on its formal interface specification. This idiom is known as duck typing, from a James Whitcomb Riley poem

Listing 1: Overloading the Print Function

```
import sys
def print(*args,sep='',end="\n",file=sys.stdout):
    __builtins__.print(*args)
    __builtins__.print(*args,file=open("log.file","a"))
```

Table 2: Strings in Python 2 and 3

	Python 2.*	Python 3.0
8-bit string	"string"	b"string"
Unicode string	u"string"	"string"

Listing 2: With Block with File

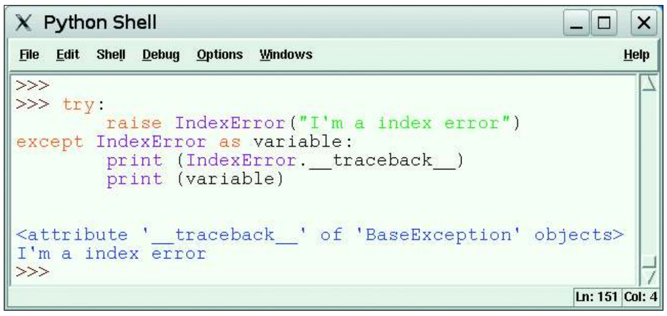
```
with open('/etc/passwd', 'r') as file:
    for line in file:
        print line,
# file is closed
```

Listing 3: Protecting a Code Block

```
with locked(myLock):
    # Code here executes with myLock held. The lock is
    # guaranteed to be released when the block is left
class locked:
    def __init__(self, lock):
        self.lock = lock
    def __enter__(self):
        self.lock.acquire()
    def __exit__(self, type, value, tb):
        self.lock.release()
```

Listing 4: Import Idiom

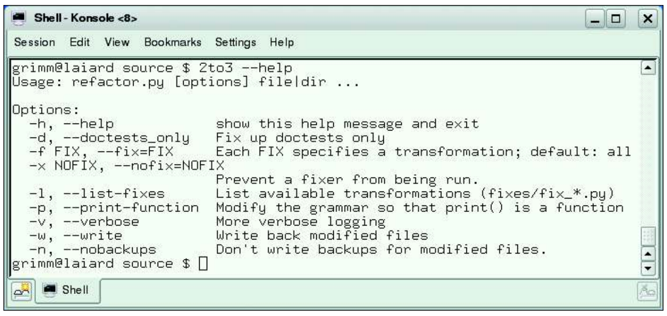
```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```



```
>>>
>>> try:
        raise IndexError("I'm a index error")
except IndexError as variable:
        print (IndexError.__traceback__)
        print (variable)


<attribute '__traceback__' of 'BaseException' objects>
I'm a index error
>>>
```

Figure 3: The new traceback attribute for exceptions.



```
grimm@laiard source $ 2to3 --help
Usage: refactor.py [options] file|dir ...

Options:
  -h, --help            show this help message and exit
  -d, --doctests_only   Fix up doctests only
  -f FIX, --fix=FIX     Each FIX specifies a transformation; default: all
  -x NOFIX, --nofix=NOFIX
                        Prevent a fixer from being run.
  -l, --list-fixes      List available transformations (fixes/fix_*.py)
  -p, --print-function  Modify the grammar so that print() is a function
  -v, --verbose         More verbose logging
  -w, --write           Write back modified files
  -n, --nobackups       Don't write backups for modified files.
grimm@laiard source $ 
```

Figure 7: The 2to3 code generator has a large number of options.

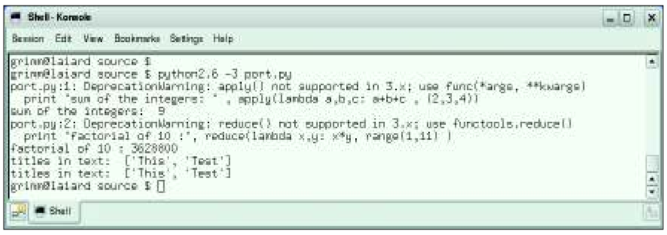
(“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”).

If a class contains an abstract method, it becomes an abstract base class and cannot be instantiated. Derivative classes can only be created if they implement these abstract methods. Abstract base classes in Python act in a similar way to abstract base classes in C++, in that abstract methods are specifically allowed to contain an implementation.

Besides abstract methods, Python also uses abstract properties. Python uses abstract base classes in the *numbers* [4] and *collections* [5] modules. That still leaves the question: When does a class become an abstract class? The class needs the *ABCMeta* metaclass. The methods can then be declared as *@abstractmethod*, and the properties as *@ abstactproperty* using the corresponding decorators. In addition to this, the use of abstract base classes means the entry of static typing into a language that uses dynamic typing. In the example in Figure 2, the Cygnus class cannot be instantiated because it does not implement the abstract *quack()* method.

Multiple Processors

Python’s response to multi-processor architectures is the new *multiprocessing* [6] library. This module imitates the well-known Python *threading* module, but instead of a thread, it creates a process, and it does this independently of the platform. The multiprocessing module became necessary because CPython, the standard Python implementation, could only run one thread in its interpreter. This behavior is dictated by the Global Interpreter Lock (GIL) [7].



```
grimm@laiard source $
grimm@laiard source $ python2.6 -3 port.py
port.py:1: DeprecationWarning: apply() not supported in 3.x; use func(*args, **kwargs)
  print "sum of the integers: " , apply(lambda a,b,c: a+b+c , (2,3,4))
sum of the integers:  9
port.py:2: DeprecationWarning: reduce() not supported in 3.x; use functools.reduce()
  print "Factorial of 10 :", reduce(lambda x,y: x*y, range(1,11) )
factorial of 10 : 3628800
titles in text:  ['This', 'Test']
titles in text:  ['This', 'Test']
grimm@laiard source $ 
```

Figure 6: The Python 3 option in v.2.6 points to issues with the port.

Class decorators [8] round out the list in Python; from now on, you can decorate classes as well as functions.

Python 3.0 also has a new I/O library [9]. The string data type gained a new *format()* [10] method for improved string formatting. At the same time, the previous format operator, %, is *deprecated* in Python 3.1.

Clean-Up Work

Wherever changes occur, it is also necessary to ditch some ballast. This affects libraries that have been removed, that have been repackaged, or that coexist in C and Python implementations. The popular Python idiom (Listing 4) of importing the fast C implementation of a module first and then falling back on the Python implementation if this fails is no longer necessary. Python does this automatically. More details are available on changes to the standard library [11].

All exceptions must derive from *BaseException*. This implies, in particular, that string exceptions are no longer supported. The exception object now has a new *__traceback__* attribute, which contains the traceback of the exception. The approach to both calling and fielding exceptions with arguments has changed. Programmers can now throw exceptions with arguments using *raise BaseException(args)* and field them with *except BaseException as variable* (Figure 3).

Python 3 also includes other changes to make life easier for programmers. For example, in cooperative *super* calls, it is no longer necessary to name the class instance and the class name. Old-style classes, which were deprecated at some previous time, no longer exist in Python 3.0; this removes the need to derive from *object* to use Python’s newer features.

Direct evaluation of input via the *input()* command is no longer sup-

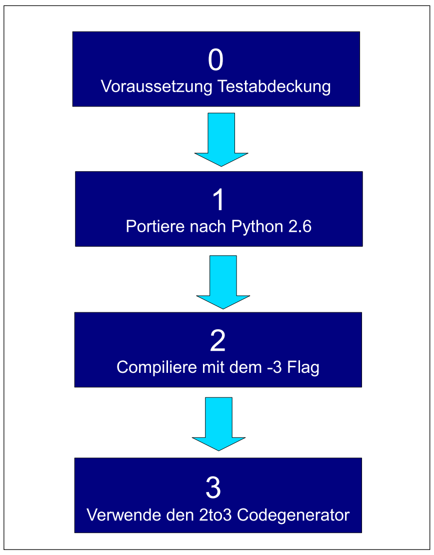
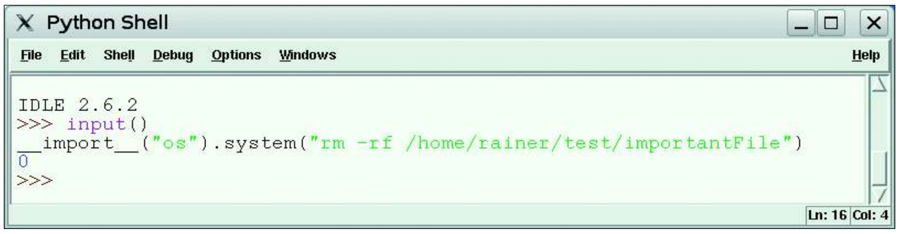

0
Voraussetzung Testabdeckung
1
Portiere nach Python 2.6
2
Compiliere mit dem -3 Flag
3
Verwende den 2to3 Codegenerator

Figure 5: The recommended migration path for porting Python 2.6 code to version 3.



```
IDLE 2.6.2
>>> input()
__import__("os").system("rm -rf /home/rainer/test/importantFile")
0
>>>
```

Figure 4: Direct evaluation of input via the input command is no longer supported in Python 3.

ported, as the input is available as an input string. This approach closes a critical security hole (Figure 4). It was only logical to rename the *raw_input()* function *input()* and to remove *raw_input*.

Of course, any description of the new features can't hope to be exclusive. If you want to know more, check out the reference document by von Rossum, "What's New In Python 3.0" [12].

Porting to Python 3.0

A clear migration path is available for porting Python 2 code to Python 3 (Figure 5), but you will need to test the code and fix any bugs at each step of the way.

The four lines of code in Listing 5 will serve as an example for migrating Python 2 to Python 3.0. All four lines defined functional components of Python. The first function calculates the sum of three numbers, 2, 3, and 4, by applying these arguments to the Lambda function. The *reduce* built-in successively reduces the list of all numbers from 1 to 10 by multiplying the results of the last multiplication with the next number in the sequence. The last two functions filter words, starting with filtering uppercase letters out of a string.

The code works on Python 2.6, and you only need to perform Steps 3 and 4 for the port. The source code for this example is stored in a file called *port.py*.

Calling the Python 2.6 interpreter with the *-3* option (Figure 6) shows incompatibilities with version 3: Both the *apply* function and the *reduce* function are no longer built-ins in Python 3.0. The code is easily fixed (Listing 6), and the deprecation warnings then stop.

The code generator *2to3* is really useful if you need to correct your Python 2 code; the generator's final step is to automatically generate code for versions 3.0 and 3.1. The tool offers several options for this (Figure 7). The direct approach is to overwrite the original file: *2to3 port.py -w*. The result is the ported source code for Python 3.0 (Listing 7).

When to Make the Move

Python 3.0 originally placed more emphasis on functionality, and this meant that it was about 10 percent slower than Python 2. The required optimization occurred in Python 3.1 [13]. This optimization relates to special handling of small integers. On top of this, Python 3.1's I/O library is implemented in C, which makes it between 2 and 20 times faster. Decoding of the UTF-8, UTF-16, and Latin-1 character sets is now twice to four times as fast.

If you are still waiting for third-party libraries to be ported, there is no point porting your application code to Python 3. von Rossum also recommends [14] not writing any code that will run on both Python 2.6 and Python 3 without modifications. It is preferable to maintain the source code as Python 2.6 code and then use automated tools to port to Python 3.0 or 3.1. Christopher Neugebauer has the final word in his video talk on Python 3000: "Learn 2.6, but keep 3k in mind." ■

Listing 5: Code for Port

```
print "sum of the integers: " , apply(lambda a,b,c: a+b+c , (2,3,4))
print "factorial of 10 :", reduce(lambda x,y: x*y, range(1,11) )
print "titles in text: ", filter( lambda word: word.istitle(),"This is a long
Test".split())
print "titles in text: ", [ word for word in "This is a long Test".split() if
word.istitle()]
```

Listing 6: Removing the Deprecation Warning

```
print "sum of the integers: " , (lambda a,b,c: a+b+c)(*(2,3,4))
import functools
print "factorial of 10 :", functools.reduce(lambda x,y: x*y, range(1,11) )
print "titles in text: ", filter( lambda word: word.istitle(),"This is a long
Test".split())
print "titles in text: ", [ word for word in "This is a long Test".split() if
word.istitle()]
```

Listing 7: Code Ported to Python 3.0

```
print("sum of the integers: " , (lambda a,b,c: a+b+c)(*(2,3,4)))
print("factorial of 10 :", reduce(lambda x,y: x*y, list(range(1,11)) ))
print("titles in text: ", [word for word in "This is a long Test".split() if
word.istitle()])
print("titles in text: ", [ word for word in "This is a long Test".split() if
word.istitle()])
```

THE AUTHOR

Rainer Grimm has been a software developer since 1999 at Science + Computing AG in Tübingen, Germany. In particular, he holds training sessions for the in-house product SC Venus.

INFO

[1] Peters, Tim. *The Zen of Python*: *http://www.python.org/dev/peps/pep-0020*

[2] *contextlib* library: *http://docs.python.org/3.0/library/contextlib.html#module-contextlib*

[3] PEP 0343: *http://www.python.org/dev/peps/pep-0343*

[4] *numbers* library: *http://docs.python.org//3.0/library/numbers.html#module-numbers*

[5] *collections* library: *http://docs.python.org/3.0/library/collections.html#module-collection*

[6] *multiprocessing* module: *http://docs.python.org/3.0/library/multiprocessing.html#module-multiprocessing*

[7] GIL: *http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock*

[8] Class decorators: *http://www.python.org/dev/peps/pep-3129*

[9] *io* library: *http://docs.python.org/3.0/library/io.html#module-io*

[10] Format string method: *http://docs.python.org/3.0/whatsnew/2.6.html#pep-3101*

[11] Changes to libraries: *http://docs.python.org/3.0/whatsnew/3.0.html#library-changes*

[12] What's New in v.3.0: *http://docs.python.org/3.0/whatsnew/3.0.html*

[13] Optimizations in v.3.1: *http://docs.python.org/dev/py3k/whatsnew/3.1.html#optimizations*

[14] Porting to v.3.0: *http://docs.python.org/3.0/whatsnew/3.0.html#miscellaneous-other-changes*