DAVID BEAZLEY

Python 3: the good, the bad, and the ugly

David Beazley is an open source software developer and the author of *Python Essential Reference* (4th edition, Addison-Wesley, 2009). He lives in Chicago, where he also teaches Python courses.

dave@dabeaz.com

IN LATE 2008, AND WITH MUCH FANFARE, Python 3.0 was released into the wild. Although there have been ongoing releases of Python in the past, Python 3 is notable in that it intentionally breaks backwards compatibility with all previous versions. If you use Python, you have undoubtedly heard that Python 3 even breaks the lowly print statement—rendering the most simple "Hello World" program incompatible. And there are many more changes, with some key differences that no conversion program can deal with. In this article I give you a taste of what's changed, outline where those changes are important, and provide you with guidance on whether you want or need to move to Python 3 soon.

By the time you're reading this, numerous articles covering all of the new features of Python 3 will have appeared. It is not my intent to simply rehash all of that material here. In fact, if you're interested in an exhaustive coverage of changes, you should consult "What's New in Python 3?" [1]. Rather, I'm hoping to go a little deeper and to explain *why* Python has been changed in the way it has, the implications for end users, and why you should care about it. This article is not meant to be a Python tutorial; a basic knowledge of Python programming is assumed.

Python's C Programming Roots

The lack of type declarations and curly braces aside, C is one of the foremost influences on Python's basic design, including the fundamental operators, identifiers, and keywords. The interpreter is written in C and even the special names such as `__init__`, `__str__`, and `__dict__` are inspired by a similar convention in the C preprocessor (for example, preprocessor macros such as `__FILE__` and `__LINE__`). The influence of C is no accident—Python was originally envisioned as a high-level language for writing system administration tools. The goal was to have a high-level language that was easy to use and that sat somewhere between C programming and the shell.

Although Python has evolved greatly since its early days, a number of C-like features have remained in the language and libraries. For example, the integer math operators are taken straight from C—even truncating division just like C:

```
>>> 7/4
1
>>>
```

Python's string formatting is modeled after the C `printf()` class of functions. For example:

```
>>> print "%10s %10d %10.2f" % ('ACME',100,123.45)
   ACME 100    123.45
>>>
```

File I/O in Python is byte-oriented and really just a thin layer over the C stdio functionality:

```
>>> f = open("data.txt","r")
>>> data = f.read(100)
>>> f.tell()
100L
>>> f.seek(500)
>>>
```

In addition, many of Python's oldest library modules provide direct access to low-level system functions that you would commonly use in C systems programs. For example, the `os` module provides almost all of the POSIX functions and other libraries provide low-level access to sockets, signals, fcntl, terminal I/O, memory mapped files, and so forth.

These aspects of Python have made it an extremely useful tool for writing all sorts of system-oriented tools and utilities, the purpose for which it was originally created and the way in which it is still used by a lot of system programmers and sysadmins. However, Python's C-like behavior has not been without its fair share of problems. For example, the truncation of integer division is a widely acknowledged source of unintended mathematical errors and the use of byte-oriented file I/O is a frequent source of confusion when working with Unicode in Internet applications.

Python 3: Breaking Free of Its Past

One of the most noticeable changes in Python 3 is a major shift away from its original roots in C and UNIX programming. Although the interpreter is still written in C, Python 3 fixes a variety of subtle design problems associated with its original implementation. For example, in Python 3, integer division now yields a floating point number:

```
>>> 7/4
1.75
>>>
```

A number of fundamental statements in the language have been changed into library functions. For example, `print` and `exec` are now just ordinary function calls. Thus, the familiar "Hello World" program becomes this:

```
print("Hello World")
```

Python 3 fully embraces Unicode text, a change that affects almost every part of the language and library. For instance, all text strings are now Unicode, as is Python source code. When you open files in text mode, Unicode is always assumed—even if you don't specify anything in the way of a specific encoding (with UTF-8 being the usual default). I'll discuss the implications of this change a little later—it's not as seamless as one might imagine.

Borrowing from Java, Python 3 takes a completely different approach to file I/O. Although you still open files using the familiar `open()` function, the kind of "file" object that you get back is now part of a layered I/O stack. For example:

```
>>> f = open("foo")
>>> f
<io.TextIOWrapper object at 0x383950>
>>>
```

So, what is this `TextIOWrapper`? It's a class that wraps a file of type `BufferedReader`, which in turns wraps a file of type `FileIO`. Yes, Python 3 has a full assortment of various I/O classes for raw I/O, buffered I/O, and text decoding that get hooked together in various configurations. Although it's not a carbon copy of what you find in Java, it has a similar flavor.

Borrowing from the .NET framework, Python 3 adopts a completely different approach to string formatting based on composite format strings. For example, here is the preferred way to format a string in Python 3:

```
print("{0:10} {1:10d} {2:10.2f}".format(name,shares,price))
```

For now, the old `printf`-style string formatting is still supported, but its future is in some doubt.

Although there are a variety of other more minor changes, experienced Python programmers coming into Python 3 may find the transition to be rather jarring. Although the core language feels about the same, the programming environment is very different from what you have used in the past. Instead of being grounded in C, Python 3 adopts many of its ideas from more modern programming languages.

Python's Evolution Into a Framework Language

If you're a current user of Python, you might have read the last section and wondered what the Python developers must be thinking. Breaking all backwards compatibility just to turn `print` into a function and make all string and I/O handling into a big Unicode playground seems like a rather extreme step to take, especially given that Python already has a rather useful `print` statement and fully capable support for Unicode. As it turns out, these changes aren't the main story of what Python 3 is all about. Let's review a bit of history.

Since its early days, Python has increasingly been used as a language for creating complex application frameworks and programming environments. Early adopters noticed that almost every aspect of the interpreter was exposed and that the language could be molded into a programming environment that was custom-tailored for specific application domains. Example frameworks include Web programming, scientific computing, image processing, and animation. The only problem was that even though Python was "good enough" to be used in this way, many of the tricks employed by framework builders pushed the language in directions that were never anticipated in its original design. In fact, there were a lot of subtle quirks, limitations, and inconsistencies. Not only that, there were completely new features that framework builders wanted—often inspired by features of other programming languages. So, as Python has evolved, it has gradually acquired a new personality.

Almost all of this development has occurred in plain sight, with new features progressively added with each release of the interpreter. Although

many users have chosen to ignore this work, it all takes center stage in Python 3. In fact, the most significant aspect of Python 3 is that it sheds a huge number of deprecated features and programming idioms in order to lay the groundwork for a whole new class of advanced programming techniques. Simply stated, there are things that can be done in Python 3 that are not at all possible in previous versions. In the next few sections, I go into more detail about this.

Python Metaprogramming

Python has always had two basic elements for organizing programs: functions and classes. A function is a sequence of statements that operate on some passed arguments and return a result. For example:

```
def factorial(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

A class is a collection of functions called methods that operate on objects known as instances. Here is a sample class definition:

```
class Rectangle(object):
    def __init__(self,width,height):
        self.width  = width
        self.height = height
    def area(self):
        return self.height*self.width
    def perimeter(self):
        return 2*self.height + 2*self.width
```

Most users of Python are familiar with the idea of *using* functions and classes to carry out various programming tasks. For example:

```
>>> print factorial(6)
720
>>> r = Rectangle(4,5)
>>> r.area()
20
>>> r.perimeter()
18
>>>
```

However, an often overlooked feature is that function and class definitions have first-class status. That is, when you define a function, you are creating a "function object" that can be passed around and manipulated just like a normal piece of data. Likewise, when you define a class, you are creating a "type object." The fact that functions and classes can be manipulated means that it is possible to write programs that carry out processing on their own internal structure. That is, you can write code that operates on function and class objects just as easily as you can write code that manipulates numbers or strings. Programming like this is known as *metaprogramming*.

METAPROGRAMMING WITH DECORATORS

A common metaprogramming example is the problem of creating various forms of function wrappers. This is typically done by writing a function that

accepts another function as input and that dynamically creates a completely new function that wraps an extra layer of logic around it. For example, this function wraps another function with a debugging layer:

```
def debugged(func):
  def call(*args,**kwargs):
    print("Calling %s" % func.__name__)
    result = func(*args,**kwargs)
    print("%s returning %r" % (func.__name__, result))
    return result
  return call
```

To use this utility function, you typically apply it to an existing function and use the result as its replacement. An example will help illustrate:

```
>>> def add(x,y):
...     return x+y
...
>>> add(3,4)
7
>>> add = debugged(add)
>>> add(3,4)
Calling add
add returning 7
7
>>>
```

This wrapping process became so common in frameworks, that Python 2.4 introduced a new syntax for it known as a *decorator*. For example, if you want to define a function with an extra wrapper added to it, you can write this:

```
@debugged
def add(x,y):
  return x+y
```

The special syntax *@name* placed before a function or method definition specifies the name of a function that will process the function object created by the function definition that follows. The value returned by the decorator function takes the place of the original definition.

There are many possible uses of decorators, but one of their more interesting qualities is that they allow function definitions to be manipulated at the time they are defined. If you put extra logic into the wrapping process, you can have programs selectively turn features on or off, much in the way that a C programmer might use the preprocessor. For example, consider this slightly modified version of the `debugged()` function:

```
import os
def debugged(func):
  # If not in debugging mode, return func unmodified
  if os.environ.get('DEBUG','FALSE') != 'TRUE':
    return func

  # Put a debugging wrapper around func
  def call(*args,**kwargs):
    print("Calling %s" % func.__name__)
    result = func(*args,**kwargs)
    print("%s returning %r" % (func.__name__, result))
    return result
  return call
```

In this modified version, the `debugged()` function looks at the setting of an environment variable and uses that to determine whether or not to put a de-

bugging wrapper around a function. If debugging is turned off, the function is simply left alone. In that case, the use of a decorator has no effect and the program runs at full speed with no extra overhead (except for the one-time call to `debugged()` when decorated functions are *defined*).

Python 3 takes the idea of function decoration to a whole new level of sophistication. Let's look at an example:

```
def positive(x):
    "must be positive"
    return x > 0

def negative(x):
    "must be negative"
    return x < 0

def foo(a:positive, b:negative) -> positive:
    return a - b
```

The first two functions, `negative()` and `positive()`, are just simple function definitions that check an input value `x` to see if it satisfies a condition and return a Boolean result. However, something very different must be going on in the definition of `foo()` that follows.

The syntax of `foo()` involves a new feature of Python 3 known as a function annotation. A function annotation is a mechanism for associating *arbitrary* values with the arguments and return of a function definition. If you look carefully at this code, you might get the impression that the `positive` and `negative` annotations to `foo()` are carrying out some kind of magic—maybe calling those functions to enforce some kind of contract or assertion. However, you are wrong. In fact, these annotations do absolutely nothing! `foo()` is just like any other Python function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
-7
>>>
```

Python 3 doesn't do anything with annotations other than store them in a dictionary. Here is how to view it:

```
>>> foo.__annotations__
{'a': <function positive at 0x384468>,
 'b': <function negative at 0x3844b0>,
 'return': <function positive at 0x384468> }
>>>
```

The interpretation and use of these annotations are left entirely unspecified. However, their real power comes into play when you mix them with decorators. For example, here is a decorator that looks at the annotations and creates a wrapper function where they turn into assertions:

```
def ensure(func):
    # Extract annotation data
    return_check = func.__annotations__.get('return',None)
    arg_checks   = [(name,func.__annotations__.get(name))
                    for name in func.__code__.co_varnames]

    # Create a wrapper that checks argument values and the return
    # result using the functions specified in annotations

    def assert_call(*args,**kwargs):
        for (name,check),value in zip(arg_checks,args):
```

```
            if check: assert check(value), "%s %s" % (name, check.__doc__)
        for name,check in arg_checks[len(args):]:
            if check: assert check(kwargs[name]), "%s %s" % (name, check.__
doc__)
        result = func(*args,**kwargs)
        assert return_check(result), "return %s" % return_check.__doc__
        return result

    return assert_call
```

This code will undoubtedly require some study, but here is how it is used in a program:

```
@ensure
def foo(a:positive, b:negative) -> positive:
    return a - b
```

Here is an example of what happens if you violate any of the conditions when calling the decorated function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
Traceback (most recent call last):
  File "", line 1, in
  File "meta.py", line 19, in call
    def assert_call(*args,**kwargs):
AssertionError: a must be positive
>>>
```

It's really important to stress that everything in this example is user-defined. Annotations can be used in any manner whatsoever—the behavior is left up to the application. In this example, we built our own support for a kind of "contract" programming where conditions can be optionally placed on function inputs. However, other possible applications might include type checking, performance optimization, data serialization, documentation, and more.

METACLASSES

The other major tool for metaprogramming is Python's support for metaclasses. When a class definition is encountered, the body of the class statement (all of the methods) populates a dictionary that later becomes part of the class object that is created. A metaclass allows programmers to insert their own custom processing into the last step of the low-level class creation process. The following example illustrates the mechanics of the "metaclass hook." You start by defining a so-called metaclass that inherits from `type` and implements a special method `__new__()`:

```
class mymeta(type):
    def __new__(cls,name,bases,dict):
        print("Creating class  :", name)
        print("Base classes   :", bases)
        print("Class body   :", dict)
        # Create the actual class object
        return type.__new__(cls,name,bases,dict)
```

Next, when defining new classes, you can add a special "metaclass" specifier like this:

```
class Rectangle(object,metaclass=mymeta):
    def __init__(self,width,height):
```

```
        self.width  = width
        self.height = height
    def area(self):
        return self.height*self.width
    def perimeter(self):
        return 2*self.height + 2*self.width
```

If you try this code, you will see the `_new__()` method of the metaclass execute once when the `Rectangle` class is *defined*. The arguments to this method contain all of the information about the class including the name, base classes, and dictionary of methods. I would strongly suggest trying this code to get a sense for what happens.

At this point, you might be asking yourself, "How would I use a feature like this?" The main power of a metaclass is that it can be used to manipulate the entire contents of class body in clever ways. For example, suppose that you collectively wanted to put a debugging wrapper around every single method of a class. Instead of manually decorating every single method, you could define a metaclass to do it like this:

```
class debugmeta(type):
    def __new__(cls,name,bases,dict):
        if os.environ.get('DEBUG','FALSE') == 'TRUE':
            # Find all callable class members and put a
            # debugging wrapper around them.
            for key,member in dict.items():
                if hasattr(member,'__call__'):
                    dict[key] = debugged(member)
        return type.__new__(cls,name,bases,dict)

class Rectangle(object,metaclass=debugmeta):
    ...
```

In this case, the metaclass iterates through the entire class dictionary and rewrites its contents (wrapping all of the function calls with an extra layer).

Metaclasses have actually been part of Python since version 2.2. However, Python 3 expands their capabilities in an entirely new direction. In past versions, a metaclass could only be used to process a class definition after the entire class body had been executed. In other words, the entire body of the class would first execute and then the metaclass processing code would run to look at the resulting dictionary. Python 3 adds the ability to carry out processing *before* any part of the class body is processed and to incrementally perform work as each method is defined. Here is an example that shows some new metaclass features of Python 3 by detecting duplicate method names in a class definition:

```
# A special dictionary that detects duplicates
class dupdict(dict):
    def __setitem__(self,name,value):
        if name in self:
            raise TypeError("%s already defined" % name)
        return dict.__setitem__(self,name,value)

# A metaclass that detects duplicates
class dupmeta(type):
    @classmethod
    def __prepare__(cls,name,bases):
        return dupdict()
```

In this example, the `__prepare__()` method of the metaclass is a special method that runs at the very beginning of the class definition. As input it receives the name of the class being defined and a tuple of base classes. It returns the dictionary object that will be used to store members of the class body. If you return a custom dictionary, you can capture each member of a class as it is defined. For example, the `dupdict` class redefines item assignment so that, if any duplicate is defined, an exception is immediately raised. To see this metaclass in action, try it with this code:

```
class Rectangle(metaclass=dupmeta):
  def __init__(self,width,height):
    self.width = self.width
    self.height = self.height
  def area(self):
    return self.width*self.height
  # This repeated method name will be rejected (a bug)
  def area(self):
    return 2*(self.width+self.height)
```

Finally, just to push all of these ideas a little bit further, Python 3 allows class definitions to be decorated. For example:

```
@foo
class Bar:
  statements
```

This syntax is shorthand for the following code:

```
class Bar:
  statements

Bar = foo(Bar)
```

So, just like functions, it is possible to use decorators to put wrappers around classes. Some possible use cases include applications related to distributed computing and components. For example, decorators could be used to create proxies, set up RPC servers, register classes with name mappers, and so forth.

HEAD EXPLOSION

If you read the last few sections and feel as though your brain is going to explode, then your understanding is probably correct. (Just for the record, metaclasses are also known as Python's "killer joke"—in reference to a Monty Python sketch that obviously can't be repeated here.) However, these new metaprogramming features are what really sets Python 3 apart from its predecessors, because these are the new parts of the language that can't be emulated in previous versions. They are also the parts of Python 3 that pave the way to entirely new types of framework development.

Python 3: The Good

On the whole, the best feature of Python 3 is that the entire language is more logically consistent, entirely customizable, and filled with advanced features that provide an almost mind-boggling amount of power to framework builders. There are fewer corner cases in the language design and a lot of warty features from past versions have been removed. For example, there are no "old-style" classes, string exceptions, or features that have plagued Python developers since its very beginning, but which could not be removed because of backwards compatibility concerns.

There are also a variety of new language features that are simply nice to use. For example, if you have used features such as list comprehensions, you know that they are a very powerful way to process data. Python 3 builds upon this and adds support for set and dictionary comprehensions. For example, here is an example of converting all keys of a dictionary to lowercase:

```
>>> data = { 'NAME' : 'Dave', 'EMAIL':'dave@dabeaz.com' }
>>> data = {k.lower():v for k,v in data.items}
>>> data
{'name': 'Dave', 'email': 'dave@dabeaz.com'}
>>>
```

Major parts of the standard library—especially those related to network programming—have been reorganized and cleaned up. For instance, instead of a half-dozen scattered modules related to the HTTP protocol, all of that functionality has been collected into an HTTP package.

The bottom line is that, as a programming language, Python 3 is very clean, very consistent, and very powerful.

Python 3: The Bad

The obvious downside to Python 3 is that it is not backwards compatible with prior versions. Even if you are aware of basic incompatibilities such as the `print` statement, this is only the tip of the iceberg. As a general rule, it is not possible to write any kind of significant program that simultaneously works in Python 2 and Python 3 without limiting your use of various language features and using a rather contorted programming style.

In addition to this, there are no magic directives, special library imports, environment variables, or command-line switches that will make Python 3 run older code or allow Python 2 to run Python 3 code. Thus, it's really important to stress that you must treat Python 3 as a completely different language, not as the next step in a gradual line of upgrades from previous versions.

The backwards incompatibility of Python 3 presents a major dilemma for users of third-party packages. Unless a third-party package has been explicitly ported to Python 3, it won't work. Moreover, many of the more significant packages have dependencies on other Python packages themselves. Ironically, it is the large frameworks (the kind of code for which Python 3 is best suited) that face the most daunting task of upgrading. As of this writing, some of the more popular frameworks aren't even compatible with Python 2.5—a release that has been out for several years. Needless to say, they're not going to work with Python 3.

Python 3 includes a tool called 2to3 that aims to assist in Python 2 to Python 3 code migration. However, it is not a silver bullet nor is it a tool that one should use lightly. In a nutshell, 2to3 will identify places in your program that might need to be fixed. For example, if you run it on a "hello world" program, you'll get this output:

```
bash-3.2$ 2to3 hello.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: ws_comma
--- hello.py (original)
+++ hello.py (refactored)
@@ -1,1 +1,1 @@
-print "hello world"
```

```
+print("hello world")
RefactoringTool: Files that need to be modified:
RefactoringTool: hello.py
bash-3.2$
```

By default, 2to3 only identifies code that needs to be fixed. As an option, it can also rewrite your source code. However, that must be approached with some caution. 2to3 might try to fix things that don't actually need to be fixed, it might break code that used to work, and it might fix things in a way that you don't like. Before using 2to3, it is highly advisable to first create a thorough set of unit tests so that you can verify that your program still works after it has been patched.

Python 3: The Ugly

By far the ugliest part of Python 3 is its revised handling of Unicode and the new I/O stack. Let's talk about the I/O stack first.

In adopting a layered approach to I/O, the entire I/O system has been reimplemented from the ground up. Unfortunately, the resulting performance is so bad as to render Python 3 unusable for I/O-intensive applications. For example, consider this simple I/O loop that reads a text file line by line:

```
for line in open("somefile.txt"):
  pass
```

This is a common programming pattern that Python 3 executes more than 40 times slower than Python 2.6! You might think that this overhead is due to Unicode decoding, but you would be wrong—if you open the file in binary mode, the performance is even worse! Numerous other problems plague the I/O stack, including excessive buffer copying and other resource utilization problems. To say that the new I/O stack is "not ready for prime time" is an understatement.

To be fair, the major issue with the I/O stack is that it is still a prototype. Large parts of it are written in Python itself, so it's no surprise that it's slow. As of this writing, there is an effort to rewrite major parts of it in C , which can only help its performance. However, it is unlikely that this effort will ever match the performance of buffered I/O from the C standard library (the basis of I/O in previous Python versions). Of course, I would love to be proven wrong.

The other problematic feature of Python 3 is its revised handling of Unicode. I say this at some risk of committing blasphemy—the fact that Python 3 treats all text strings as Unicode is billed as one of its most important features. However, this change now means that Unicode pervades every part of the interpreter and its standard libraries. This includes such mundane things as command-line options, environment variables, filenames, and low-level system calls. It also means that the intrinsic complexity of Unicode handling is now forced on *all* users regardless of whether or not they actually need to use it.

To be sure, Unicode is a critically important aspect of modern applications. However, by implicitly treating all text as Unicode, Python 3 introduces a whole new class of unusual programming issues not seen in previous Python versions. Most of these problems stem from what remains a fairly loose notion of any sort of standardized Unicode encoding in many systems. Thus, there is now a potential mismatch between low-level system interfaces and the Python interpreter.

To give an example of the minefield awaiting Python 3 users, let's look at a simple example involving the file system on a Linux system. Take a look at this directory listing:

```
% ls -b
image.png jalape\361o.txt readme.txt
%
```

In this directory, there is a filename Jalepe\361o.txt with an extended Latin character, "ñ," embedded in it. Admittedly, that's not the most common kind of filename one encounters on a day-to-day basis, but Linux allowed such a filename to be created, so it must be assumed to be technically valid.

Past versions of Python have no trouble dealing with such files, but let's take a look at what happens in Python 3. First, you will notice that there is no apparent way to open the file:

```
>>> f = open("jalape\361o.txt")
Traceback (most recent call last):
IOError: [Errno 2] No such file or directory: 'jalapeño.txt'
>>>
```

Not only that, the file doesn't show up in directory listings or with file globbing operations—so now we have a file that's invisible! Let's hope that this program isn't doing anything critical such as making a backup.

```
>>> os.listdir(".")
['image.png', 'readme.txt']
>>> glob.glob("*.txt")
['readme.txt']
>>>
```

Let's try passing the filename into a Python 3.0 program as a command-line argument:

```
% python3.0 *.txt
Could not convert argument 2 to string
%
```

Here the interpreter won't run at all—end of story.

The source of these problems is the fact that the filename is not properly encoded as UTF-8 (the usual default assumed by Python). Since the name can't be decoded, the interpreter either silently rejects it or refuses to run at all. There are some settings that can be made to change the encoding rules for certain parts of the interpreter. For example, you can use `sys.setfilesystemencoding()` to change the default encoding used for names on the file system. However, this can only be used after a program starts and does not solve the problem of passing command-line options.

Python 3 doesn't abandon byte-strings entirely. Reading data with binary file modes [e.g., `open(filename,"rb")`] produces data as byte-strings. There is also a special syntax for writing out byte-string literals. For example:

```
s = b'Hello World'          # String of 8-bit characters
```

It's subtle, but supplying byte-strings is one workaround for dealing with funny file names in our example. For example:

```
>>> f = open(b'jalape\361o.txt')
>>> os.listdir(b'.')
[b'jalape\xf1o', b'image.png', b'readme.txt']
>>>
```

Unlike past versions of Python, byte-strings and Unicode strings are strictly separated from each other. Attempts to mix them in any way now produce an exception:

```
>>> s = b'Hello'
>>> t = "World"
>>> s+t Traceback (most recent call last):
  File "", line 1, in
TypeError: can't concat bytes to str
>>>
```

This behavior addresses a problematic aspect of Python 2 where Unicode strings and byte-strings could just be mixed together by implicitly promoting the byte-string to Unicode (something that sometimes led to all sorts of bizarre programming errors and I/O issues). It should be noted that the fact that string types can't be mixed is one of the most likely things to break programs migrated from Python 2. Sadly, it's also one feature that the 2to3 tool can't detect or correct. (Now would be a good time to start writing unit tests.)

System programmers might be inclined to use byte-strings in order to avoid any perceived overhead associated with Unicode text. However, if you try to do this, you will find that these new byte-strings do not work at all like byte-strings in prior Python versions. For example, the indexing operator now returns byte values as integers:

```
>>> s[2]
108
>>>
```

If you print a byte-string, the output always includes quotes and the leading b prefix—rendering the `print()` statement utterly useless for output except for debugging. For example:

```
>>> print(s)
b'Hello'
>>>
```

If you write a byte-string to any of the standard I/O streams, it fails:

```
>>> sys.stdout.write(s)
Traceback (most recent call last):
  File "", line 1, in
  File "/tmp/lib/python3.0/io.py", line 1484, in write
    s.__class__.__name__)
TypeError: can't write bytes to text stream
>>>
```

You also can't perform any kind of meaningful formatting with byte-strings:

```
>>> b'Your age is %d' % 42
Traceback (most recent call last):
  File "", line 1, in
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'
>>>
```

Common sorts of string operations on bytes often produce very cryptic error messages:

```
>>> 'Hell' in s
Traceback (most recent call last):
  File "", line 1, in
TypeError: Type str doesn't support the buffer API
>>>
```

This does work if you remember that you're working with byte-strings:

```
>>> b'Hell' in s
True
>>>
```

The bottom line is that Unicode is something that you will be forced to embrace in Python 3 migration. Although Python 3 corrects a variety of problematic aspects of Unicode from past versions, it introduces an entirely new set of problems to worry about, especially if you are writing programs that need to work with byte-oriented data. It may just be the case that there is no good way to entirely handle the complexity of Unicode—you'll just have to choose a Python version based on the nature of the problems you're willing to live with.

Conclusions

At this point, Python 3 can really only be considered to be an initial prototype. It would be a mistake to start using it as a production-ready replacement for Python 2 or to install it as an "upgrade" to a Python 2 installation (especially since no Python 2 code is likely to work with it).

The embrace of Python 3 is also by no means a foregone conclusion in the Python community. To be sure, there are some very interesting features of Python 3, but Python 2 is already quite capable, providing every feature of Python 3 except for some of its advanced features related to metaprogramming. It is highly debatable whether programmers are going to upgrade based solely on features such as Unicode handling—something that Python already supports quite well despite some regrettable design warts.

The lack of third-party modules for Python 3 also presents a major challenge. Most users will probably view Python 3 as nothing more than a curiosity until the most popular modules and frameworks make the migration. Of course this raises the question of whether or not the developers of these frameworks see a Python 3 migration as a worthwhile effort.

It will be interesting to see what programmers do with some of the more advanced aspects of Python 3. Many of the metaprogramming features such as annotations present interesting new opportunities. However, I have to admit that I sometimes wonder whether these features have made Python 3 too clever for its own good. Only time will tell.

General Advice

For now, the best advice is to simply sit back and watch what happens with Python 3—at the very least it will be an interesting case study in software engineering. New versions of Python 2 continue to be released and there are no immediate plans to abandon support for that branch. Should you decide to install Python 3, it can be done side-by-side with an existing Python 2 installation. Unless you instruct the installation process explicitly, Python 3 will not be installed as the default.

REFERENCES

[1] What's New in Python 3?: http://docs.python.org/dev/3.0/whatsnew/3.0.html.