LINUX VOICE
TUTORIAL

# PYTHON: MIGRATE YOUR PROGRAMS TO VERSION 3

RICHY DELANEY

It's been long enough: it's time to port your applications to Python 3 and take advantage of its great new features.

**WHY DO THIS?**

- Embrace the future!
- Take advantage of the latest improvements to Python.
- Get users around the world with Unicode characters.

Python, like most other pieces of software, sees incremental releases fairly frequently. If you have used Python on any Linux distribution in the past decade, it's likely that you were using Python 2 of some variety (Python 2.0 was released in October 2000, and the last Python release of the 2.x versions was 2.7).

All of the previous releases of Python have been backwards compatible, which means that although the underlying code has changed with each release, the core structure of Python remained the same. The syntax as well as the majority of the standard library APIs were unchanged. This meant that code that runs on Python 2.5 for example should run on Python 2.7.

Python 3 was the first release of Python that was backwards incompatible. This led to a slow uptake, and many developers were reluctant to put in the extra effort to update their software. There is still a massive subset of the Python community who use Python 2.X rather than the new version.

However, there were also advantages to breaking compatibility with Python 2. It has enabled the developers to get rid of a lot of the cruft that had built up in the language, and helped them to add some modern features. Mistakes in the core API can be corrected without having to worry about maintaining compatible with a whole host of older software.

## The language evolves

It's important to understand how Python is developed and designed by its community. Thanks to the very open design discussions through Python Enhancement Proposals (see the *What In The World Are PEPs* boxout, right), there are reasoned discussions around every change in the language. Decisions are made on these discussions by the Python community at large and of course by the Benevolent Dictator for Life (BDFL) Guido Van Rossum, the original creator of the Python programming language. Along with this form of discussion, there are also a lot of language decisions made on the Python development mailing list.

Most of the decision making and design proposals for the Python 3 took place during PEPs 3000 and 3100, which are still online for all to see. It is through this open forum that the Python language is developed and is truly one of the great parts of the language and its ecosystem. What makes Python 3 worth the switch? Why should your new project be written in Python 3? For many years following the release of Python 3, the lack of quality external libraries meant that it was often not a good choice. Nowadays, this has changed significantly, and the number of libraries that support Python 3 is growing every day. Major Python projects such as the Django web framework now support Python 3 on its stable branch.

Coupled with this, Python 2.X is now an old language that, although maintained and supported, is no longer developed. Guido recently extended the life of Python 2.7 to 2020, but Python 2.7 is a language which is guaranteed to not have any new features in its lifespan. With Python 3, you get all the latest and greatest features in the standard library.

## Fantastic features

If you read about Python 3 online, you may be forced into thinking the only difference is that the print statement is now a function, but Python 3 also adds real improvements:

- Unicode is supported throughout the standard library and is the default type for any strings defined.
- The new asyncio library, which is part of the standard library, gives a defined way to do asynchronous programming in Python. This makes it easy to write concurrent programs enabling you to make the most of your new-generation hardware.
- Better exception handling: in Python 2.X there were lots of ways to throw and catch exceptions; with Python 3, error handling is cleaner and improved.
- Virtualenv is now part of the standard Python distribution. The Virtualenv environment provides

### What in the world are PEPs?

PEPs (Python Enhancement Proposals) are the main forums in the Python community for proposing new features or improvements to the Python core language. They enable the community to review, discuss and improve proposals. They also enable readers to digest why certain features are the way they are as well as looking at the alternates were rejected and why, so they're often helpful in understanding how you would use a certain feature – a good example being PEP-8, (**http://python.org/dev/peps/pep-0008**), which documents the suggested coding style for Python. Popular tools such as pep8 and flake8 enforce these rules when run on a Python file. The main PEP index can be found at **http://python.org/dev/peps**.

a way to build a lightweight Python runtime that enables you to have isolated Python environments running potentially different Python versions and libraries. Previously this was third-party software which had to be installed separately. If Python 3 is installed on your system, then so is Virtualenv.

- PYC (Python Byte Compiled) files now live in a new directory called **pycache**. In previous Python releases, PYC files cluttered directories as they lived in the same place as the source files. There is also an improvement in how they are stored, as there are separate files per interpreter.
- There is now a single number type in Python. Prior to Python 3, there were two types: **longs** and **int**, which has been simplified to just the **int** type.
- The standard library itself is much improved in lots of places.

The main thing to note is that the effect of the changes in Python 3 on the syntax and overall usability of the language is very small. This should mean that existing Python developers feel very comfortable with Python 3 and that Python 3 is just as user friendly as other Python versions are to new users of the language.

At the time of writing, mainstreams distributions do not ship Python 3 as the default distribution. However, some systems come with Python 3 installed, while others have it in the repositories. Ubuntu comes with Python 3 pre-installed in 14.04. In 12.04, you can install Python 3 by getting the **python3-minimal** package from your package manager. Ubuntu and Fedora have open tickets to make Python 3 the default Python. This is currently scheduled for Fedora 22, and was an unreached goal for Ubuntu 14.04. More information on installing Python 3 on Ubuntu can be found in the boxout below.

## Porting to Python 3

Let's take a look at what's involved in porting some code from Python 2.7 to 3.3. This example will explore some of the main pain points and improvements in Python 3.3 – the code is quite simple, but in the wild there are a lot of different challenges that you could face when trying to port large bodies of code. All of the code listings, for Python 2.7 and Python 3 versions, can be found at **linuxvoice.com/lv5-python.tar.gz**.

The code we're using is a simple locale-specific calculator program. This program displays some of the inherent problems with the way that Python 2.X manages Unicode. Below is a full code listing of the code written for Python 2.7 :

```
#!/usr/bin/python2.7
# -*- coding: utf-8 -*-
import argparse

ENGLISH_MESSAGES = {
  "greeting": "Hello world",
  "num1": "First Number:",
  "num2": "Second Number:",
  "div": "Dividing %d by %d gives %f"
}
CHINESE_MESSAGES = {
  "greeting": u"世界，你好！",
  "num1": u"第一个号码".encode("utf-8"),
  "num2": u"第二个号码".encode("utf-8"),

  "div": u"除以%d%d给出了%f".encode("utf-8")
}

LOCALES = {
  "en": ENGLISH_MESSAGES,
  "ch": CHINESE_MESSAGES
}

def grab_input(locale_messages, locale_key):
  user_msg = locale_messages[locale_key]
  x = raw_input(user_msg)
  # python 2.X converts this automatically to a long
  # in the case where it is > sys.maxint
  # something you don't have to think about with
  # python 3
  return int(x)

def div(x, y, locale_messages):
  # need to first make x a float before we can divide
  floated_x = float(x)
  print locale_messages["div"] % (
    x, y, (floated_x / y))
  return floated_x / y
if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("--locale", "-l", dest="locale",
default="en")
  args = parser.parse_args()
  locale_messages = LOCALES[args.locale]
  print locale_messages["greeting"]
  x = grab_input(locale_messages, "num1")
  y = grab_input(locale_messages, "num2")

  div(x, y, locale_messages)
```

This program asks a user for two numbers, and then displays the result of addition, subtraction and division of these numbers. The main difficulty in this program is the Unicode support for the Chinese locale. The following is a demo run of the program using the Chinese locale.

```
$ python2.7 simple_calc.py --locale=ch
世界，你好！
```

### Installing Python 3 from PPA

If you're using an older version of Ubuntu, the version of Python 3 installed may be as old as Python 3.2 (in the case of Ubuntu 12.04). Luckily, there is a PPA called "deadsnakes" maintained by Felix Krull. At time of writing, this supported the following Python revisions: 2.3, 2.4, 2.5, 2.6, 2.7, 3.1, 3.2, 3.3, 3.4. The PPA can be added by adding the line **ppa:fkrull/deadsnakes** to your software sources. After the PPA is configured on your system you can install Python 3.3, for example, with:

```
sudo apt-get install python3.3
```

```
第一个号码27
第二个号码4
加入 27 ~ 4 给了我 31
减去27至4给了我23
除以274给出了6.750000
```

Let's dig a little deeper into the portions of the code that deal with Unicode.

```
# -*- coding: utf-8 -*-
```

This line tells the Python interpreter which encoding it should use to read the Python file. Without this comment, Python defaults to reading the file using the ASCII encoding. This encoding enables us to read Unicode literals in the source code such as the Chinese messages in the greetings dictionary. Omitting this from our source code will result in the following exception when running the code.

```
SyntaxError: Non-ASCII character '\xe4' in file simple_calc.py on line 14, but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

The next thing to understand in the source is how the Chinese characters are declared.

```
"num1": u"第一个号码".encode("utf-8"),
```

The unicode literal is encoded using the **"utf-8"** encoding. This converts from a Unicode type to a **str** type in Python, which is what Python 2.7 uses by default. If we don't encode the Unicode literal, the Python **raw_input** function will not be able to output a Unicode literal and execution will fail as in the following example:

```
>>> unicode_value = u"世界，你好！"
>>> raw_input(unicode_value)
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5: ordinal not in range(128)
```

This is one of the problems in Python 2.7: the default string encoding is ASCII, meaning that any non-ASCII encoded strings must be encoded to work in some core functions like **raw_input**.

## Integer division

Another Python 2.7-specific functionality that has been replaced in Python 3.3 is integer division. This behaviour is summed up in the two examples below: one is code running in a Python 2.7 interpreter, while the other is running in a Python 3 interpreter.

Python 2.7:

```
>>> x, y = 21, 5
>>> x / y
4
>>> float(x) / y
4.2
>>> x / float(y)
4.2
```

Python 3.3:

```
>>> x, y = 21, 5
>>> x / y
4.2
4
```

As should be clear from the examples, in Python 2.7 division of two integers always resulted in an integer. In Python 3.3, division of two integers will always result in a float value. The previous default behaviour is preserved in Python 3.3 under the **//** operator.

As mentioned before, porting a big body of code is very different to our example and there are a few things that we have to make sure of before attempting any port to Python 3.

- The project is thoroughly unit-tested. Regressions and bugs in the porting process can easily go unnoticed without a full suite of tests.
- The project is using Python 2.7. Using the latest version of the Python 2.X interpreter allows the developer to benefit from some features that are backported from Python 3 and is the easiest migration path.

To make porting code to Python 3 as easy as possible, there's a tool called **2to3**, which should be on the system path if Python 3 is installed on your system. **2to3** is a utility script, which takes Python 2.7 file(s) as input and outputs the equivalent code compatible with the Python 3 interpreter.

First, before making any modifications, the **2to3** tool enables you to output a diff of what modifications will be done:

```
2to3 simple_calc.py
```

For the sake of this tutorial, we want to preserve the Python 2.7 version of the code and create a new version of the file in a directory called **python3**. Here's the command to output a Python 3 version of the code into the **python3** directory (presuming that the Python 2.7 version of the file is housed under the **python27** directory):

```
2to3 -o python3 -nw python27/simple_calc.py
```

The **-o** argument specifies the directory to write to and the **-nw** tells the tool there's no need to write any backups (it's worth a look at the man page for a comprehensive guide to the options in **2to3**). This command will output a file called **simple_calc.py** in the **python3** directory.

Let's look at the changes that the **2to3** code made and what changes are needed to make it work to our expectations.

The first major change that happens is one of the most controversial and best-known features of Python 3. Printing output in Python 2.7 was done using a statement rather than a function, ie no parenthesis. In Python 3 this was replaced by a function. This backwards-incompatible change is one of the far-reaching changes, and makes running Python 2.X code in a Python 3 interpreter often impossible without using a tool like **2to3**. Below is an example of the change made by **2to3**.

Python 2.7:

```
print locale_messages["greeting"]
```

Python 3.3:

```
print(locale_messages["greeting"])
```

There are a number of reasons to have **print** as a function. A function enables you to more easily override and add functionality to **print** without having to change Python syntax. It also makes it easier to mock out in unit tests. The next thing of note is that

**2to3** replaced the **raw_input** function for retrieving user input with the simpler function name of **input**. Here's the Python 2.7 code:

```
x = raw_input(user_msg)
```

... and here's the equivalent in Python 3.3:

```
x = input(user_msg)
```

Finally, in Python 3, string literals' default type is Unicode. This means that we no longer have to include the **u** before the string literal to indicate that it is a unicode string. This preceding **u** was actually removed as part of Python 3, but to help with migration to Python 3, was re-included (as documented in PEP 414). Unicode is everywhere in Python 3 by default. The **2to3** tool has stripped the Chinese strings of the preceding **u**.

Running the generated code using Python 3 produces output similar to the following:

```
$ python3 python3/simple_calc.py --locale=ch
世界， 你好！
b'\xe7\xac\xac\xe4\xb8\x80\xe4\xb8\xaa\xe5\x8f\xb7\xe7\
xa0\x81'456
b'\xe7\xac\xac\xe4\xba\x8c\xe4\xb8\xaa\xe5\x8f\xb7\xe7\
xa0\x81'32
Traceback (most recent call last):
  File "python3/simple_calc.py", line 68, in <module>
    add(x, y, locale_messages)
  File "python3/simple_calc.py", line 42, in add
    x, y, (x + y)))
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'
```

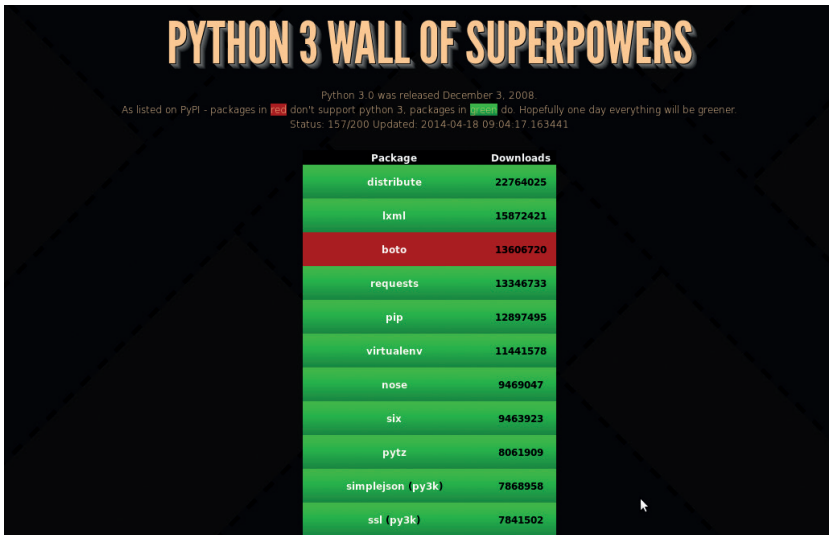
## There is no magic wand!

This example demonstrates the fact that although in most cases the **2to3** utility can make Python 2.7 code runnable, it doesn't necessarily fix any logical issues or discrepancies that result from the porting exercise. The converted program has a number of runtime errors that are both visible and cause program execution to fail. These cases highlight the necessity of having a full suite of unit tests before attempting to port code.

The main problem exhibited while running the code under Python 3 is that we were previously converting the Unicode Chinese characters to **str** types. This meant that when we convert it, the **str** type becomes the **bytes** type, which is rendered in its full representation (including the **b**). As Unicode is now the default string type in Python, there is no need to first encode it to **"utf-8"**. The Python 3 interpreter example below demonstrates this:

```
>>> x = u"第一个号码"
>>> print(x)
第一个号码
>>> print(x.encode("utf-8"))
b'\xe7\xac\xac\xe4\xb8\x80\xe4\xb8\xaa\xe5\x8f\xb7\xe7\
xa0\x81'
```

To fix this, remove the encodings in the **CHINESE_MESSAGES** dictionary declaration. The **input** function in Python 3 accepts Unicode strings as the input string, whereas the **raw_input** for Python 2 expects strings only.




Python Wall of Superpowers is a colourful way to check for Python 3 support in external libraries.

The program failed with a Type Error exception. This is due to the fact that string formatting using the **%** symbol in Python 3 does not work on bytes strings. In order to use this string formatting, the messages must be Unicode strings. The fix employed in the last step will also fix this, as the strings being operated on will now be of the Unicode type. Bytestrings will support **%** symbol formatting soon, and details of this change can be found in PEP 461.

In the last step of this conversion, Python 3's new division operator, can save a few lines of code. For the division function, we were previously casting **int**s to floats in order to get floating number results from the division. In Python 3, this is the default behaviour of division for two **int** values, meaning there is no need to cast to a float before that point. The modified **div** function is shown below:

```
def div(x, y, locale_messages):
  print(locale_messages["div"] % (
    x, y, (x / y)))
  return x / y
```

Porting even this very simple application took some effort after use of the automatic tool to make it work as expected. Larger bodies of code will result in significantly more work.

There is huge momentum currently in the Python community around improving the library support for Python 3, and there are many resources to give you an indication of what libraries are supported by Python 3. The Python 3 Wall of Superpowers website (**http://python3wos.appspot.com**) shows an up-to-date listing of the compatibility of major Python Package Index libraries with Python 3. If your project uses a lot of external libraries, this is a great way to evaluate if porting your code is possible. If you have the expertise, and a library you depend on doesn't support Python 3, why not take it on as a personal challenge and submit a pull request for that project?

**Richy Delaney is a software engineer with Demonware Ireland, working on back-end web services using Python and Linux. He has been an avid Linux user for the past five years.**