Python 2 vs Python 3

Let's start by looking at a regular class definition. Then we'll add super using Python 2 to see how it changes.

```
class MyParentClass(object):
    def __init__(self):
        pass

class SubClass(MyParentClass):
    def __init__(self):
        MyParentClass.__init__(self)
```

This is a pretty standard set up for single inheritance. We have a base class and then the subclass. Another name for base class is parent class or even super class. Anyway, in the subclass we need to initialize the parent class too. The core developers of Python thought it would be a good idea to make this kind of thing more abstract and portable, so the super function was added. In Python 2, the subclass would look like this:

```
class SubClass(MyParentClass):
    def __init__(self):
        super(SubClass, self).__init__()
```

Python 3 simplified this a bit. Let's take a look:

```
class MyParentClass():
    def __init__(self):
        pass

class SubClass(MyParentClass):
    def __init__(self):
        super().__init__()
```

The first change you will notice is that the parent class no longer needs to be explicitly based on the **object** base class. The second change is the call to **super**. We no longer need to pass it anything and yet super does the right

thing implicitly. Most classes actually have arguments passed to them though, so let's look at how the super signature changes in that case:

```
class MyParentClass():
    def __init__(self, x, y):
        pass

class SubClass(MyParentClass):
    def __init__(self, x, y):
        super().__init__(x, y)
```

Here we just need to call the super's **__init__** method and pass the arguments along. It's still nice and straight-forward.