ANNEXURE

3

Comparison between Python 2.x and Python 3.x Versions

In this annexure, we will discuss some of the important differences between Python 2.x and Python 3.x versions with examples.

Division operator: When we execute a code written in Python 3.x on a machine that has Python 2.x installed, then the results of integer division will vary (no error is raised). You must use the floating value (like `7.0/5` or `7/5.0`) to get the expected result when importing the code.

```
#In Python 2.x
>>> print 7/5
1
>>> print -7/5
-2
```

```
#In Python 3.x
>>> print(7/5)
1.4
>>> print(-7/5)
-1.4
```

`print` statement and `print()` function: While you have the `print` statement in Python 2.x, it is replaced by `print` function in Python 3.x.

```
#In Python 2.x
>>> print "Hello World"
Hello World
```

```
#In Python 3.x
>>> print("Hello World")
Hello World
```

Parsing user inputs: In Python 2.x, we use `raw_input()` function to accept user's input but in Python 3.x, the `input()` function is used. While `raw_input()` reads the input as string, the `input()` reads it as string object.

```
#In Python 2.x
num = raw_input("Enter a number : ")
print type(num)
OUTPUT
Enter a number : 5
<type 'str'>
```

```
#In Python 3.x
num = input("Enter a number : ")
print(type(num))
OUTPUT
Enter a number : 5
<class 'str'>
```

Unicode: In Python 2, implicit `str` type is `ASCII`, but in Python 3.x, it is Unicode. Python 2.x has no byte type but supports `bytearray` to a certain extent. However, Python 3.x supports Unicode (utf-8) strings and 2 byte classes, namely, `byte` and `bytearrays`.

```
#In Python 2.x
>>>print type('default string ')
<type 'str'>
```

```
#In Python 3.x
>>>print(type('default string '))
<class 'str'>
```

```
>>>print type(u'unicode string ')
<type 'unicode'>
>>>print type(b'byte string')
<type 'str'>
>>>print type(bytearray'bytearray')
<type 'bytearray'>
```

```
>>>print(type(u'string with b '))
<class 'str'>
>>>print(type(b'byte string'))
<class 'byte'>
>>>print(type(bytearray'bytearray'))
<class 'bytearray'>
```

Note Unicode strings are more versatile than ASCII strings. They can store letters from forei languages as well as emojis and the standard Roman letters and numerals.

xrange() and range(): The xrange() of Python 2.x does not exist in Python 3.x. While the rang returns a list, i.e. , range(5) returns [0, 1, 2, 3, 4], the xrange() returns an object, i.e., xrange(5) retu an iterator object which would generate a number when needed. The main difference between range() xrange() is that range() provides a static list and xrange() reconstructs the sequence every time. Althou xrange() does not support slices and other list methods, it saves memory when the task is to iterate ove large range.

xrange() is generally faster if you have to iterate over all the elements only once (example, in a loop). But, in case you need to repeat the iteration multiple times, range() performs better. In Python the range() function does all the work that xrange() function does in Python 2.x.

```
#In Python 2.x
for I in xrange(1,5):
        print I,
OUTPUT
1   2   3   4
```

```
#In Python 3.x
for I in xrange(1,5):
        print(I,end=' ')
OUTPUT
NameError: name 'xrange' is not defined
```

There is one more point of dissimilarity in Python 2.x and Python 3.x with respect to the range() functi In Python 3.x, range objects have a new method called the __contains__ method which was not present Python 2.x. The __contains__ method can speed-up the "look-ups" in Python 3.x significantly for integ and Boolean types.

Comparing unorderable types: Another good change in Python 3.x is that a TypeError is raised a warning, if we try to compare unorderable types.

```
#In Python 2.x
print "[10, 20] > 'HELLO' = ", [10, 20] >
'HELLO'
print "(10, 20) > 'HELLO' = ", (10, 20) >
'HELLO'
print "[10, 20] > (10, 20) = ", [10, 20] >
(10, 20)
OUTPUT
[10, 20] > 'HELLO' =  False
(10, 20) > 'HELLO' =  True
[10, 20] > (10, 20) =  False
```

```
#In Python 3.x
print("[10, 20] > 'HELLO' = ", [10, 20] >
'HELLO')
print("(10, 20) > 'HELLO' = ", (10, 20) >
'HELLO')
print("[10, 20] > (10, 20) = ", [10, 20]
(10, 20))
OUTPUT
TypeError: unorderable types: list() >
str()
```

Error Handling: While handling error handling Python 3.x requires the as keyword. However, as is n required in Python 2.x.

```
#In Python 2.x
try:
    print i
```

```
#In Python 3.x
try:
    print(i)
```

```
except NameError:
    print 'Name Error Generated'
OUTPUT
Name Error Generated
```

```
except NameError:
    print('Name Error Generated')
OUTPUT
name 'i' is not defined Name Error Generated
```

Raising exceptions: While in Python 2.x, you can raise an exception with or without using parentheses. In Python 3.x, it is mandatory to enclose the exception argument in parentheses otherwise, a syntax error will be generated.

```
#In Python 2.x
>>> raise IOError, "My Error"

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise IOError, "My Error"
IOError: My Error
```

```
#In Python 3.x
>>> raise IOError("My Error")

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise IOError, "My Error"
IOError: My Error
```

The next() function and .next() method: Python 2.x supports both next() and .next() which are used to iterate over the elements. But Python 3.x does not support .next(). Therefore, calling the .next() raises an AttributeError.

```
#In Python 2.x
my_generator = (letter for letter in
'Hello WOrld !!!')
print next(my_generator)
print my_generator.next()
OUTPUT
H
e
```

```
#In Python 3.x
my_generator = (letter for letter in
'Hello WOrld !!!')
print(next(my_generator))
print(my_generator.next())
OUTPUT
H
AttributeError
```

Returning iterable objects instead of lists: The range() function in Python 2.x returns a list but the same function in Python 3.x returns an iterable object.

```
#In Python 2.x
print type(range(5))
OUTPUT
<type 'list'>
```

```
#In Python 3.x
print(type(range(5)))
OUTPUT
<class 'list'>
```

Other functions and methods that do not return lists in Python 3.x include zip(), map(), filter(), dictionary's.keys() method, dictionary's.values() method, and dictionary's.items() method.

__future__ module: This __future__ module is imported in a program to use Python 3.x features in Python 2.x code. Basically, it is not a difference between Python 2.x and 3.x but a bridge that helps you incorporate new features in your code written in Python 2.x.

```
from __future__ import division
print 18 / 5
print -12 / 5
OUTPUT
3.6
-2.4
```