

Introduction to unicode

One of the major changes in Python 3 was the move to make all strings Unicode. Previously, there was a **str** type and a **unicode** type. For example:

```
# Python 2
x = 'blah'
print (type(x))
#str

y = u'blah'
print (type(y))
#unicode
```



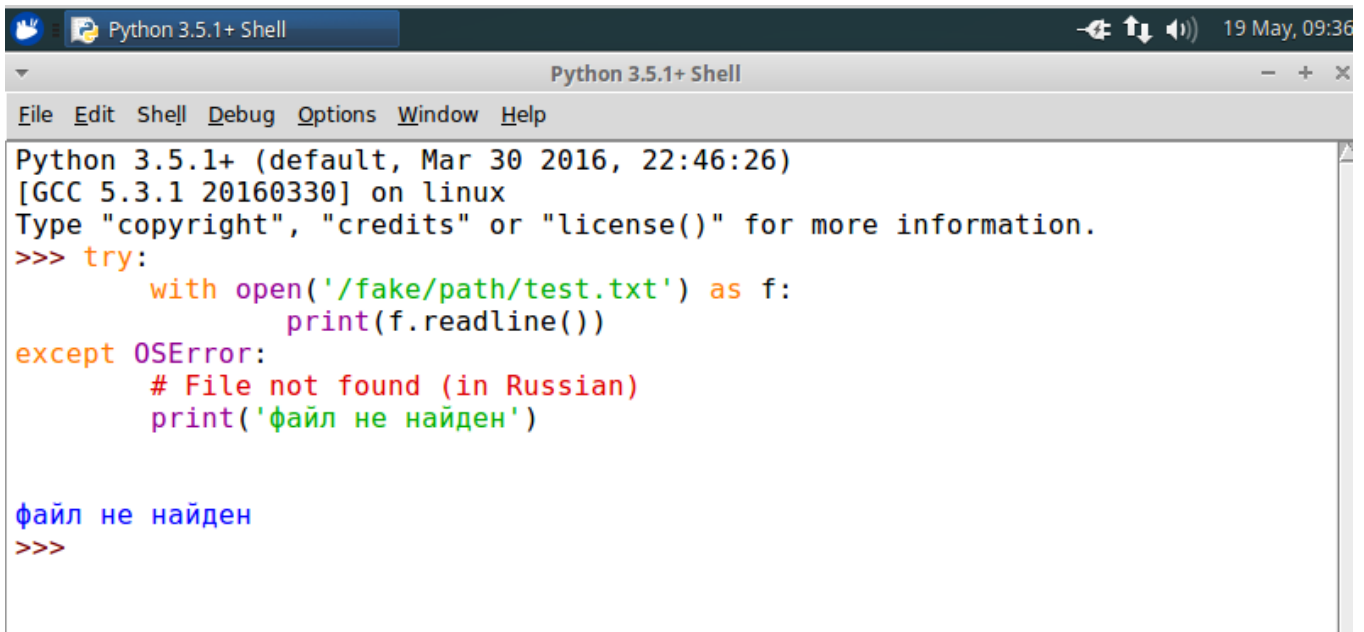
If we do the same thing in Python 3, you will note that it always returns a string type:

```
# Python 3
x = 'blah'
print (type(x))
#<class 'str'>

y = u'blah'
print (type(y) )
#<class 'str'>
```



Python 3 defaults to the UTF-8 encoding. What all this means is that you can now use Unicode characters in your strings and for variable names. Let's see how this works in practice:

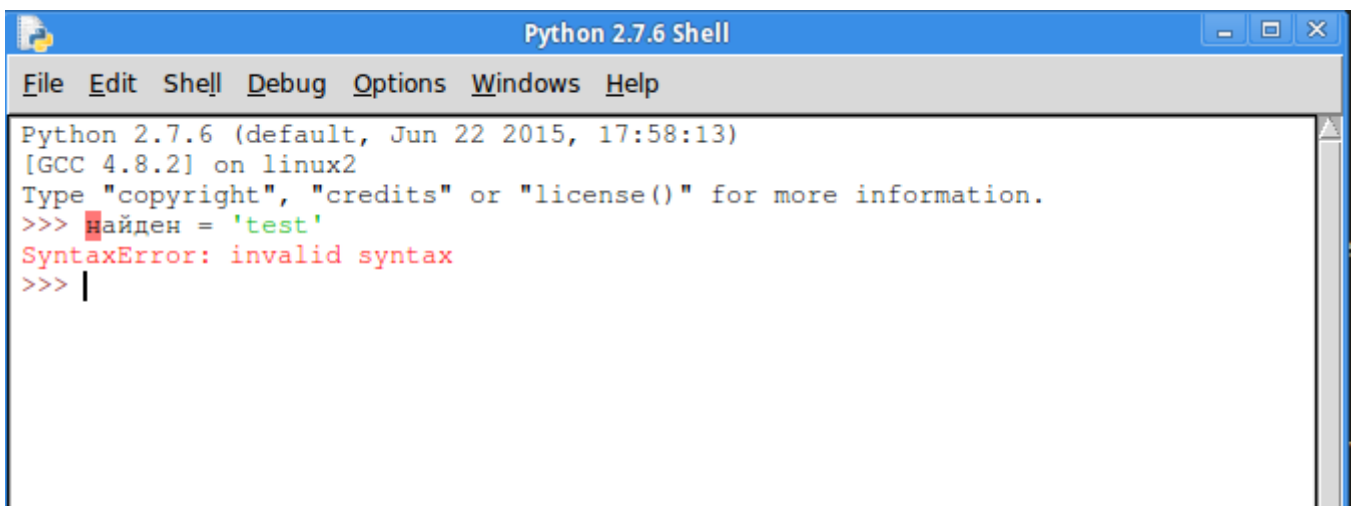


```
Python 3.5.1+ Shell
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
Type "copyright", "credits" or "license()" for more information.
>>> try:
        with open('/fake/path/test.txt') as f:
            print(f.readline())
except OSError:
    # File not found (in Russian)
    print('файл не найден')

файл не найден
>>>
```

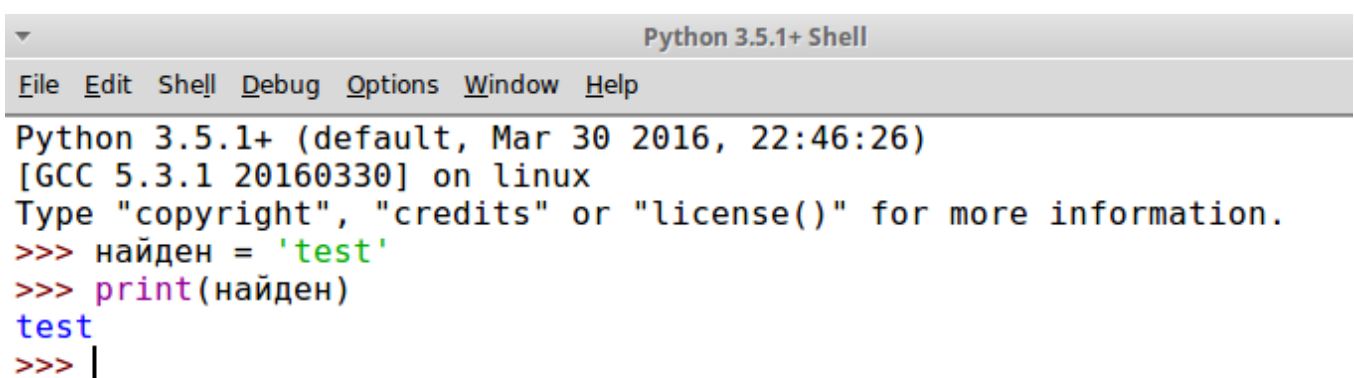
For this example, we tried to open a file that didn't exist and pretended we were in Russia. When the file wasn't found, we caught the error and printed out the error message in Russian. Note that I used Google Translate for this example, so the wording might be a little off.

Let's try creating a unicode variable name in Python 2:



```
Python 2.7.6 Shell
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> найден = 'test'
SyntaxError: invalid syntax
>>> |
```

Python 2 will throw a **SyntaxError** when we try to use Unicode in our variable name. Now let's see how Python 3 handles it:



```
Python 3.5.1+ Shell
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
Type "copyright", "credits" or "license()" for more information.
>>> найден = 'test'
>>> print(найден)
test
>>> |
```

Unicode variable names work just fine in the latest Python. In Python 2, I was always running into oddball issues when I would read a file or web page that wasn't in ASCII. An example that you might see in your output might look something like this:

```
# Python 2
print ('abcdef' + chr(255))
#'abcdef\xff'
```



Note that the end of that string has some funny characters there. That should be a “ÿ” instead of `xff`, which is basically a hex version of the character. In Python 3 you will get what you expect:

```
# Python 3
print (('abcdef' + chr(255)).encode('utf-8'))
#b'abcdef\xc3\xbf'
```



One thing I was always tempted to try to fix this issue in Python 2 was to wrap the call in Python's built-in unicode function. It's supposed to cast a string to Unicode after all. What could go wrong? Let's find out!

```
# Python 2
unicode('abcdef' + chr(255))

#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 2, in <module>
#    unicode('abcdef' + chr(255))
#UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6: ordinal not in range
```



The **UnicodeDecodeError** exception can be a royal pain in Python 2. I know I spent hours fighting it on certain projects. I look forward to not having to deal with those kinds of issues so much in Python 3. I will note that there is a

handy package on the Python Packaging Index called Unidecode that can take most Unicode characters and turn them into ASCII. I have resorted to using that in the past to fix certain issues with input.