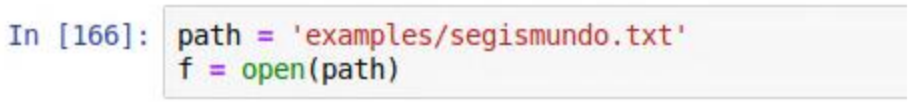
Python Built-in Data Structures, Functions, and Files

Part 7

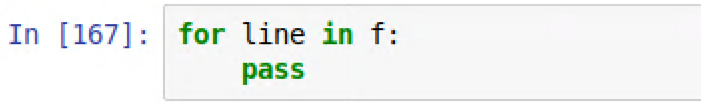
Files and the Operating System

- To open a file for reading or writing, use the built-in open function with either a relative or absolute file path:



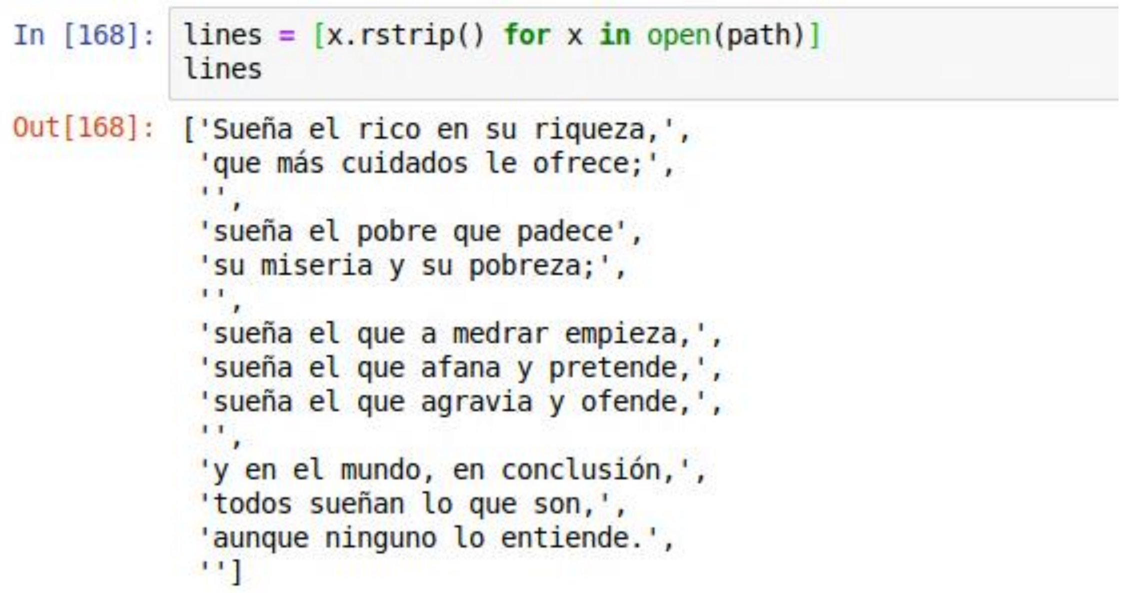
```
In [166]: path = 'examples/segismundo.txt'
          f = open(path)
```

- By default, the file is opened in read-only mode `'r'`.
- We can then treat the file handle `f` like a list and iterate over the lines like so:



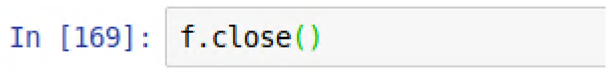
```
In [167]: for line in f:
              pass
```

- The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like:



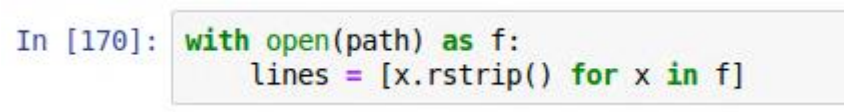
```
In [168]: lines = [x.rstrip() for x in open(path)]
          lines

Out[168]: ['Sueña el rico en su riqueza,',
           'que más cuidados le ofrece;',
           '',
           'sueña el pobre que padece',
           'su miseria y su pobreza;',
           '',
           'sueña el que a medrar empieza,',
           'sueña el que afana y pretende,',
           'sueña el que agravia y ofende,',
           '',
           'y en el mundo, en conclusión,',
           'todos sueñan lo que son,',
           'aunque ninguno lo entiende.',
           '']
```

- When you use open to create file objects, it is important to explicitly close the file when you are finished with it.
- Closing the file releases its resources back to the operating system:



```
In [169]: f.close()
```

- One of the ways to make it easier to clean up open files is to use the `with` statement:



```
In [170]: with open(path) as f:
              lines = [x.rstrip() for x in f]
```

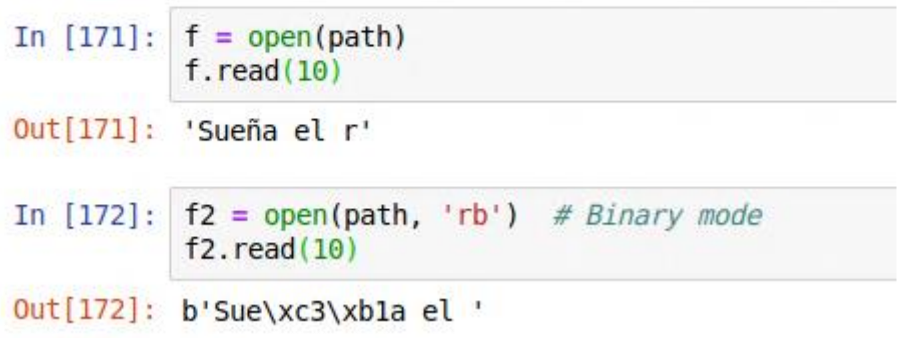
- This will automatically close the file `f` when exiting the with block.

- If we had typed `f = open(path, 'w')`, a new file at `examples/segismundo.txt` would have been created (be careful!), overwriting any one in its place.
- There is also the `'x'` file mode, which creates a writable file but fails if the file path already exists.

Mode	Description
`r`	Read-only mode
`w`	Write-only mode; creates a new file (erasing the data for any file with the same name)
`x`	Write-only mode; creates a new file, but fails if the file path already exists

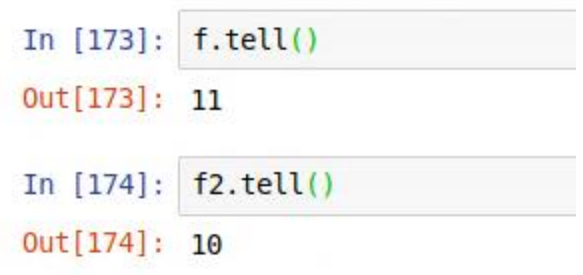
`a`	Append to existing file (create the file if it does not already exist)
`r+`	Read and write
`b`	Add to mode for binary files (i.e., `'rb'` or `'wb'`)
`t`	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add `t` to other modes to use this (i.e., `'rt'` or `'xt'`)

- For readable files, some of the most commonly used methods are `read`, `seek`, and `tell`.
- `read` returns a certain number of characters from the file.
- What constitutes a “character” is determined by the file’s encoding (e.g., UTF-8) or simply raw bytes if the file is opened in binary mode:



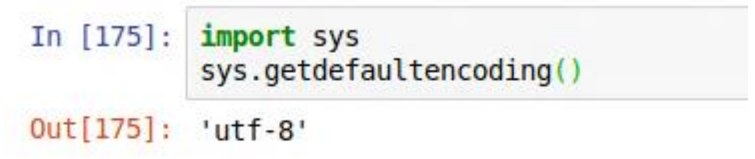
```
In [171]: f = open(path)
          f.read(10)

Out[171]: 'Sueña el r'

In [172]: f2 = open(path, 'rb')  # Binary mode
          f2.read(10)

Out[172]: b'Sue\xc3\xb1a el '
```

- The `read` method advances the file handle's position by the number of bytes read.
- `tell` gives you the current position:



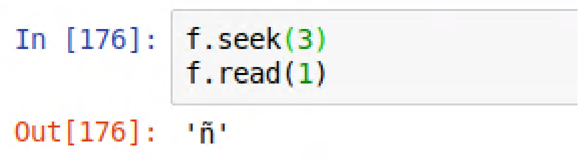
```
In [173]: f.tell()
Out[173]: 11

In [174]: f2.tell()
Out[174]: 10
```

- Even though we read 10 characters from the file, the position is 11 because it took that many bytes to decode 10 characters using the default encoding.
- You can check the default encoding in the `sys` module:



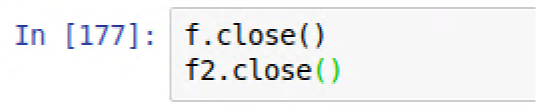
```
In [175]: import sys
          sys.getdefaultencoding()

Out[175]: 'utf-8'
```

- `seek` changes the file position to the indicated byte in the file:



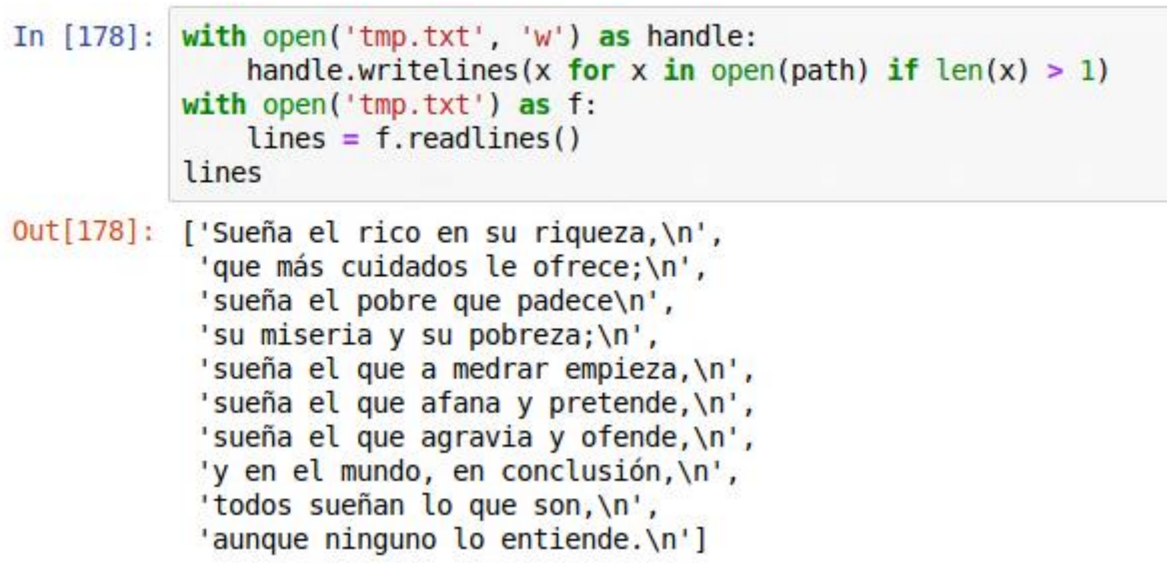
```
In [176]: f.seek(3)
          f.read(1)
Out[176]: 'ñ'
```

- Lastly, we remember to close the files:



```
In [177]: f.close()
          f2.close()
```

- To write text to a file, you can use the file's `write` or `writelines` methods.
- For example, we could create a version of `prof_mod.py` with no blank lines like so:

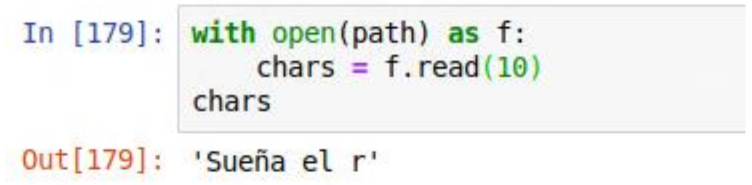


```
In [178]: with open('tmp.txt', 'w') as handle:
              handle.writelines(x for x in open(path) if len(x) > 1)
          with open('tmp.txt') as f:
              lines = f.readlines()
          lines

Out[178]: ['Sueña el rico en su riqueza,\n',
           'que más cuidados le ofrece;\n',
           'sueña el pobre que padece\n',
           'su miseria y su pobreza;\n',
           'sueña el que a medrar empieza,\n',
           'sueña el que afana y pretende,\n',
           'sueña el que agravia y ofende,\n',
           'y en el mundo, en conclusión,\n',
           'todos sueñan lo que son,\n',
           'aunque ninguno lo entiende.\n']
```

Bytes and Unicode with Files

- The default behavior for Python files (whether readable or writable) is *text mode*, which means that you intend to work with Python strings (i.e., Unicode).
- This contrasts with *binary mode*, which you can obtain by appending `b` onto the file mode.

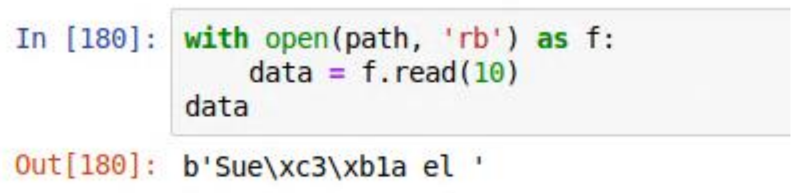
- Let’s look at the file (which contains non-ASCII characters with UTF-8 encoding) from the previous section:



```
In [179]: with open(path) as f:
              chars = f.read(10)
          chars

Out[179]: 'Sueña el r'
```

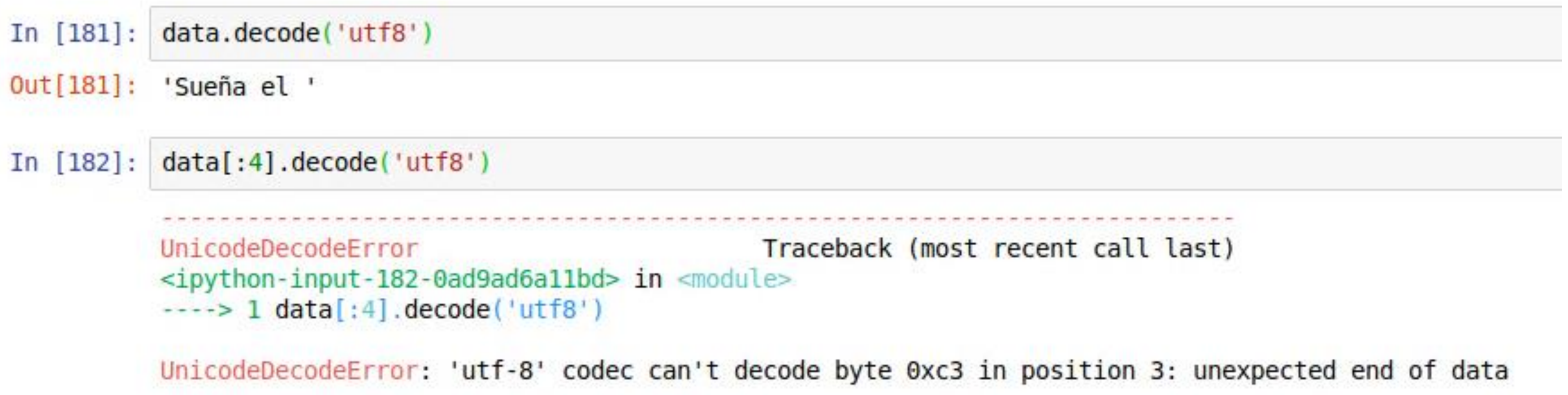
- UTF-8 is a variable-length Unicode encoding, so when I requested some number of characters from the file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from the file to decode that many characters.

- If I open the file in `'rb'` mode instead, read requests exact numbers of bytes:



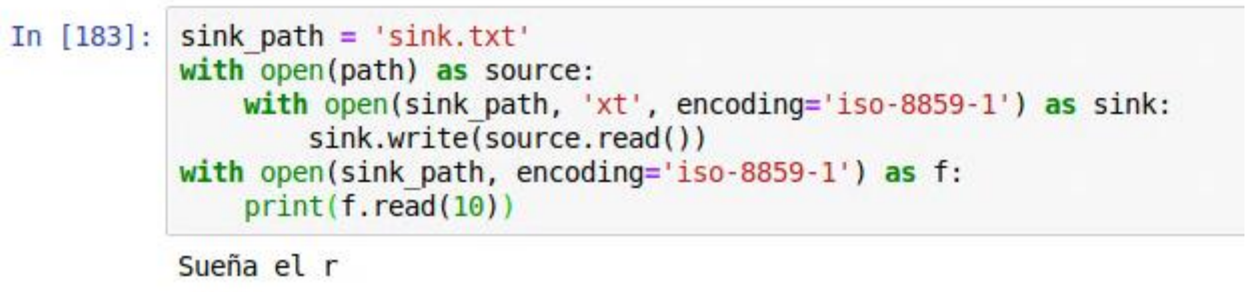
```
In [180]: with open(path, 'rb') as f:
              data = f.read(10)
          data

Out[180]: b'Sue\xc3\xb1a el '
```

- Depending on the text encoding, you may be able to decode the bytes to a `str` object yourself, but only if each of the encoded Unicode characters is fully formed:



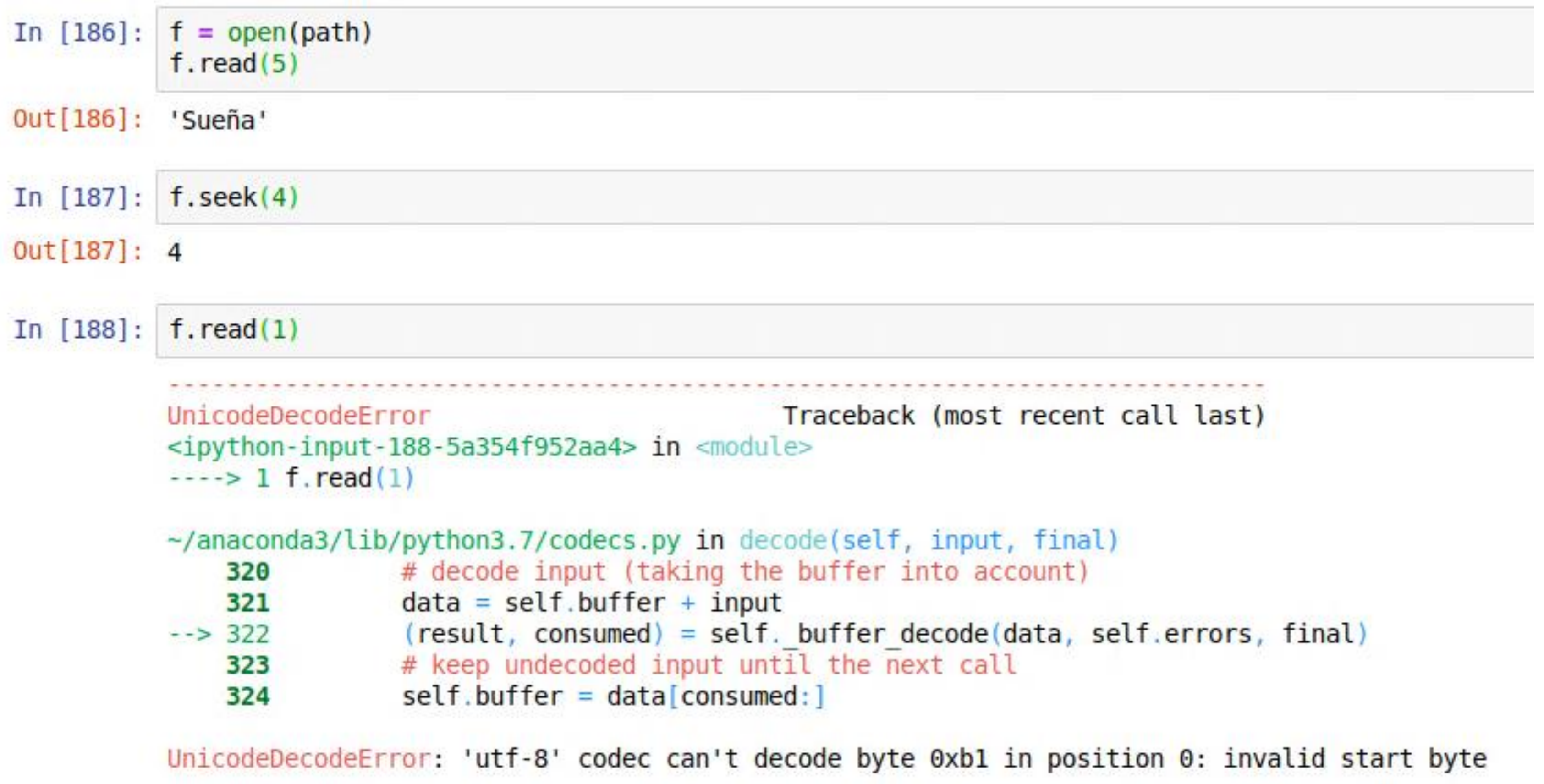
```
In [181]: data.decode('utf8')

Out[181]: 'Sueña el '

In [182]: data[:4].decode('utf8')

---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-182-0ad9ad6a11bd> in <module>
----> 1 data[:4].decode('utf8')

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpected end of data
```

- Text mode, combined with the `encoding` option of `open`, provides a convenient way to convert from one Unicode encoding to another:



```
In [183]: sink_path = 'sink.txt'
          with open(path) as source:
              with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
                  sink.write(source.read())
          with open(sink_path, encoding='iso-8859-1') as f:
              print(f.read(10))

          Sueña el r
```

- Beware using `seek` when opening files in any mode other than binary.
- If the file position falls in the middle of the bytes defining a Unicode character, then subsequent reads will result in an error:



```
In [186]: f = open(path)
          f.read(5)

Out[186]: 'Sueña'

In [187]: f.seek(4)

Out[187]: 4

In [188]: f.read(1)

          ---------------------------------------------------------------------------
          UnicodeDecodeError                        Traceback (most recent call last)
          <ipython-input-188-5a354f952aa4> in <module>
          ----> 1 f.read(1)

          ~/anaconda3/lib/python3.7/codecs.py in decode(self, input, final)
              320         # decode input (taking the buffer into account)
              321         data = self.buffer + input
          --> 322         (result, consumed) = self._buffer_decode(data, self.errors, final)
              323         # keep undecoded input until the next call
              324         self.buffer = data[consumed:]

          UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid start byte
```

- If you find yourself regularly doing data analysis on non-ASCII text data, mastering Python’s Unicode functionality will prove valuable.