File I/O

Global Knowledge®

Objectives

- See how to read and write multiple file types
- Use deterministic cleanup for files correctly
- Work with in-memory streaming APIs
- Work with paths and directories cross-platform

File I/O in Python

- There are five common types of file operations
  - **Text**
    - Read / write text of any format
    - String-based IO
  - **Binary**
    - Read / write binary of any format
    - Stream bytes and bytearray in and out
  - **XML**
    - Load XML documents
    - Parse / query documents using XPath
  - **JSON**
    - Convert JSON to / from dictionaries
    - Convert JSON to / from custom classes
  - **Pickling**
    - Serialize object graphs to proprietary binary formats

Text I/O [modes]

- Opening and creating text files
  - Uses `open(filename, mode)` built-in

Mode	Meaning
r	Open text file for **reading**. Stream is positioned at the **beginning** of the file.
r+	Open for **reading** and **writing**. The stream is positioned at the **beginning** of the file.
w	**Truncate** file to zero length or create text file for writing. The stream is positioned at the **beginning** of the file.
w+	Open for **reading** and **writing**. The file is **created** if it does not exist, otherwise it is **truncated**. The stream is positioned at the **beginning** of the file.
a	Open for **writing**. The file is **created** if it does not exist. The stream is positioned at the **end** of the file.
a+	Open for **reading** and **writing**. The file is **created** if it does not exist. The stream is positioned at the **end** of the file.

Text I/O [reading examples]

Open file with built-in open method.

Utility methods make text files easy.

```
csvFileName = "SomeData.csv"

fin = open(csvFileName, 'r', encoding="utf-8")
lines = fin.readlines()
```

Loads all data at once

Text file handles are iterable (line by line)

```
csvFileName = "SomeData.csv"

fin = open(csvFileName, 'r', encoding="utf-8")
for line in fin:
    print(line, end='')
```

Uses deferred iteration

Text I/O [cleaning up]

Files should be closed ASAP.

```
csvFileName = "SomeData.csv"

fin = open(csvFileName, 'r', encoding="utf-8")
lines = fin.readlines()
fin.close()
```

```
csvFileName = "SomeData.csv"

with open(csvFileName, 'r', encoding="utf-8") as fin:
    for line in fin:
        print(line, end='')
```

The **with** statement makes this trivial, even in the case of exceptions or early returns.

Text I/O [writing text files]

Create or open text file for appending with **a+** mode

```
with open("app.log", 'a+', encoding="utf-8") as fout:
    fout.write("The application is starting up...\n")
    fout.write("Everything looks good.\n")
```

Write method takes a string, appends it to the file

There is no ‘writeline’ but you can make one.

Text I/O [in-memory stream - reading]

The **io** package has helpful utility classes

We can treat this string as an incoming text-based file stream

```
import io

txt = """\
This is my text.
There are many words like it
But this one is my own\
"""
fin = io.StringIO(txt)

for l in fin:
    print(l, end='')

# prints
# This is my text.
# There are many words like it
# But this one is my own
```

Text I/O [in-memory stream - writing]

The **io** package has helpful utility classes

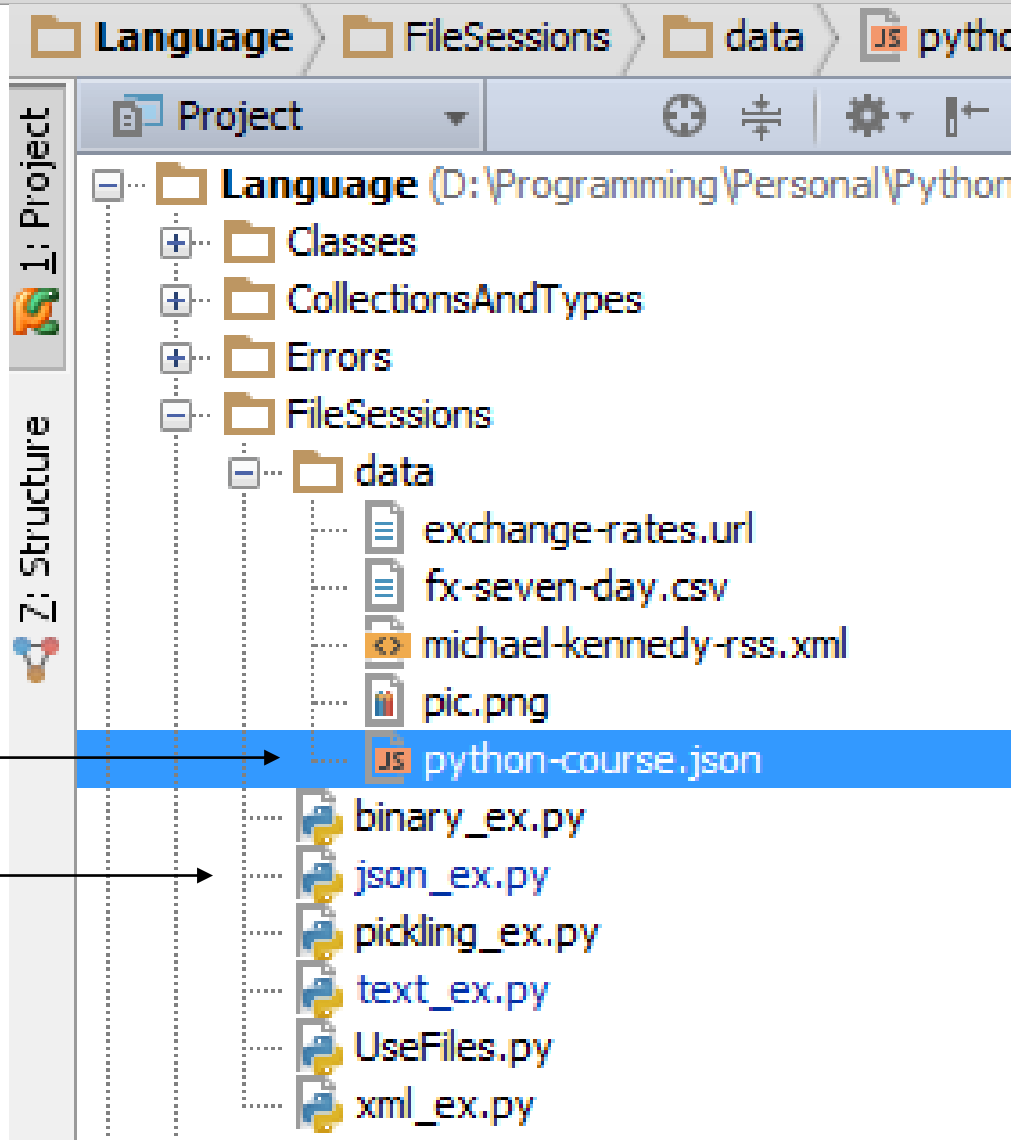
We can treat this in-memory stream as a text file handle, mode w+.

```
import io

fout = io.StringIO()

fout.write("This is line one!\n")
fout.write("This is line two!\n")

fout.seek(0)
print(fout.read())

# prints
# This is line one!
# This is line two!
```

Working with file paths (cross-platform)

How do I locate this file

From this file?



Note: this must be cross-platform safe.

Working with file paths (cross-platform)

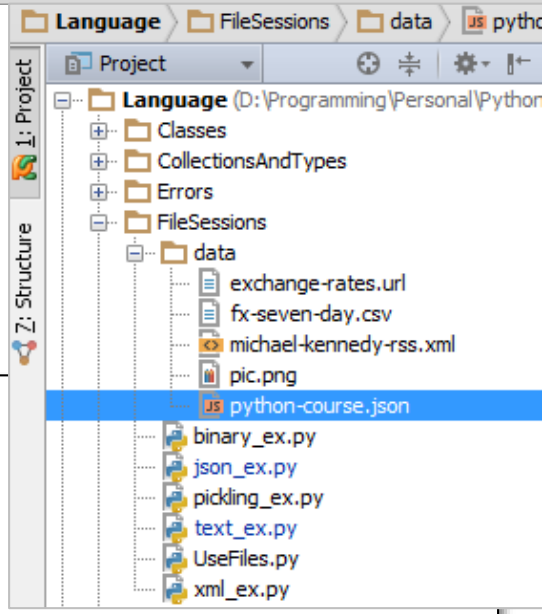
OS module has path and file tools

os.path.dirname() gets the folder

__file__ is the script

os.path.join() creates the new file path

```
import os

srcFile = __file__
srcDir = os.path.dirname(srcFile)
file = 'python-course.json'

targetFile = os.path.join(srcDir, 'data', file)

print(targetFile)

# prints this on OS X
#/Users/mkennedy/epython/Language/FileSessions/data/python-course.json

# prints this on Windows
#D:\Python_Course\Language\FileSessions\data\python-course.json
```



Binary I/O [reading files]

Incoming data can be stored in **bytearray** or directly processed.

Must specify **binary mode** (**rb**)

```
bytes = bytearray()

with open(srcFile, 'rb') as fin:
    chunkSize = 1024
    buffer = fin.read(chunkSize)
    while buffer:
        bytes.extend(buffer)
        buffer = fin.read(chunkSize)
```

Read buffer sized chunks and store or process them.

Binary I/O [writing files]

Use mode **'wb'**

```
memStream = getBinaryDataToSave()

# iteratively write (buffered)
with open(destFile, 'wb') as fout:
    for b in memStream:
        fout.write(b)

# write all in one shot
with open(destFile, 'wb') as fout:
    allBytes = memStream.getbuffer()
    fout.write(allBytes)
```

Write byte by byte

Memory streams have a simpler method

XML Files

- XML file support is built-in to Python
  - Import the **`xml.etree`** module
  - The **ElementTree** XML API provides simple DOM-based API

Import **xml.etree** module

Load xml via files

Load xml via strings

```
from xml.etree import ElementTree

xmlFile = "blog.rss.xml"
dom = ElementTree.parse(xmlFile)

xmlContent = "<rss><channel>...</channel></rss>"
dom = ElementTree.fromstring(xmlContent)
```

XML Files [querying data]

- Given this RSS data, find all titles and related links.

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
    <channel>
        <title>Michael Kennedy on Technology</title>
        <link>http://blog.michaelckennedy.net</link>
        <item>
            <title>Watch Building beautiful web...</title>
            <link>http://blog.michaelckennedy.net/...</link>
        </item>
        <item>
            <title>MongoDB for .NET developers</title>
            <link>http://blog.michaelckennedy.net/...</link>
        </item>
        <item>...</item>
     </channel>
</rss>
```

XML Files [querying data]

Search for elements using dom.**findall**()

Extract the data from each item

```
from xml.etree import ElementTree
dom = ElementTree.parse("blog.rss.xml")

items = dom.findall('channel/item')
print("Found {0} blog entries.".format(len(items)))

entries = []
for item in items:
    title = item.find('title').text
    link  = item.find('link').text
    entries.append( (title, link) )
```

```
Found 50 blog entries.
entries[:3] =>
[
   ('title1', 'link1'),
   ('title2', 'link2'),
   ('title3', 'link3),
]
```

JSON data

- JSON support comes built-in to Python
  - import the **`json`** module
  - serialize dictionaries
  - serialize custom objects
    - that have been built to support JSON
    - that do not intentionally support JSON

JSON data [parsing JSON]

- Python dictionaries' and JSON string representations are extremely similar.
  - Converting between them should be easy

Python dictionary

```
{
   'hobbies': [
      'biking',
      'motocross',
      'hiking'],
    'name': 'Michael',
    'email': '...'
}
```

JSON string

```
{
  "hobbies": [
       "biking",
       "motocross",
       "hiking"],
   "email": "...",
   "name": "Michael"
}
```

JSON data [JSON to dictionaries]

Access JSON classes
by importing **json**

Start with JSON
text as a string

```
import json

jsonTxt = '{"name": "Jeff", "age": 24}'

d = json.loads(jsonTxt)
print( type(d) )
print( d )

# prints:
# <class 'dict'>
# {'age': 24, 'name': 'Jeff'}
```

json.loads converts a
string to a dictionary

Note: **json.load** converts a file to a dictionary (pass a file **stream** as the parameter).

JSON data [dictionaries to JSON]

Access JSON classes
by importing **json**

Start with a
dictionary

```
import json

d = dict(name="Jeff", age=24)

jsonTxt = json.dumps(d)
print( jsonTxt )

# prints: '{"age": 24, "name": "Jeff"}'
```

json.dumps converts
a dictionary to a string

Note: **json.dump** converts a dictionary to a file.

JSON data [objects to JSON]

- Classes cannot be directly converted to JSON

```
class Person(object):
    def __init__(self, name, hobbies, email):
        self.name = name
        self.email = email
        self.hobbies = hobbies
```

Not supported

```
import json

jeff = Person('Jeff', [], 'j@develop.com')
jsonTxt = json.dumps(jeff)

# TypeError:
<Person object at 0x000000000250CEF0> is not JSON serializable
```

JSON data [objects to JSON]

- Classes cannot be directly converted to JSON
  - But their dictionaries can be
  - Converting back is harder

```
class Person(object):
    def __init__(self, name, hobbies, email):
        self.name = name
        self.email = email
        self.hobbies = hobbies
```

```
import json

jeff = Person('Jeff', [], 'j@develop.com')
jsonTxt = json.dumps(jeff.__dict__)
print(jsonTxt)

# prints:
# {"hobbies": [], "email": "j@develop.com", "name": "Jeff"}
```

JSON data [objects to JSON]

- Adding JSON support to our class

```
import json

class Person(object):
    def toJSON(self):
        return json.dumps(self.__dict__)
```

```
jeff = Person('Jeff', [], 'j@develop.com')
jsonTxt = jeff.toJSON()
print(jsonTxt)

# prints:
# {"hobbies": [], "email": "j@develop.com", "name": "Jeff"}
```

JSON data [JSON to objects]

- Adding JSON **parsing** support to our class

```
import json

class Person(object):
    def toJSON(self): ...

    @staticmethod
    def fromJSON(jsonText):
        d = json.loads(jsonText)
       return Person(**d) # requires arg names to match
```

```
jsonTxt = '{"hobbies": [], "email": "j@develop.com", "name": "Jeff"}'

jeff = Person.fromJSON(jsonTxt)
type(jeff) # <class Person>
```

JSON data [for humans]

- For nested data, indentation can be a big help

```
import json

d = dict(name="Jeff", age=24, hobbies=['skiing', 'hiking'])

jsonTxt = json.dumps(d, indent=4)
print( jsonTxt )

# prints:
{
    "age": 24,
    "hobbies": [
        "skiing",
        "hiking"
    ],
    "name": "Jeff"
}
```

Binary object serialization

- Python supports a proprietary binary serialization format
  - Called Pickle
  - Good for short-term storage

dump writes object to binary file

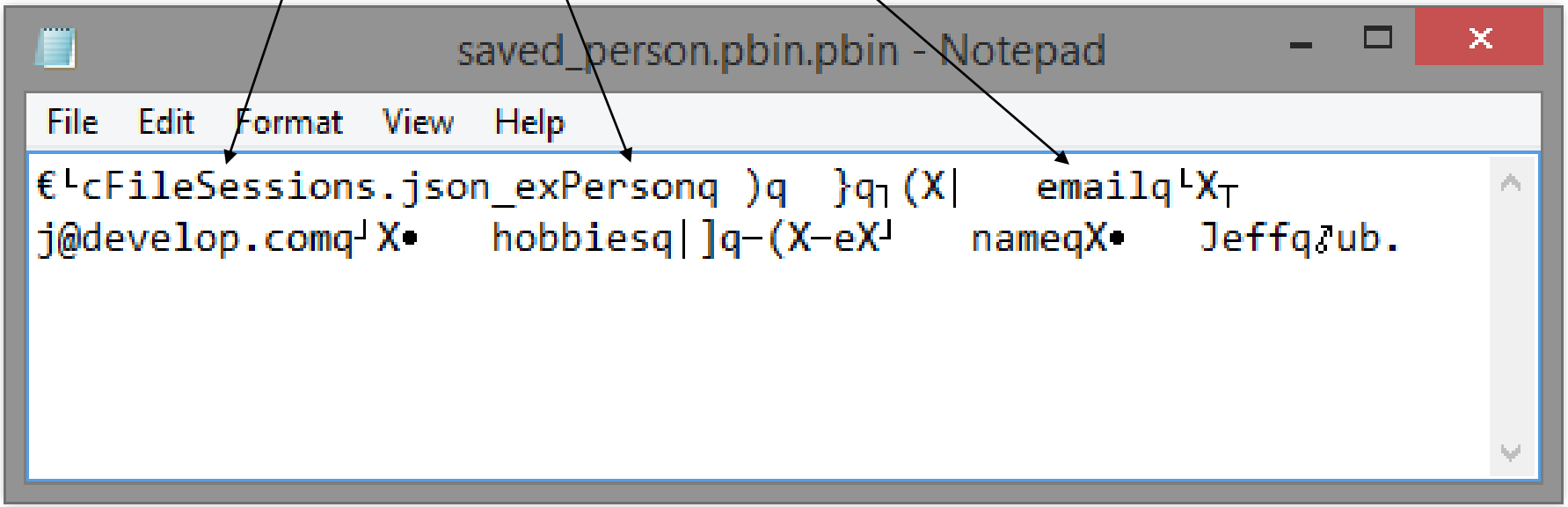
```
import pickle

jeff = Person('Jeff', [], 'j@develop.com')
pFile = 'saved_person.pbin'

with open(pFile, 'wb+') as fout:
    pickle.dump(jeff, fout)

with open(pFile, 'rb') as fin:
    newJeff = pickle.load(fin)
```

load reads object from binary file

Binary object serialization [limitations]

- Picking is not good for
  - Code that may change (fields, module names, class names)

Many module, class, and field details are hard coded in the file format



Binary object serialization [security]

- Unpickling can result in arbitrary code execution
  - Do not use pickle files for IPC with untrusted clients / services
  - Trusted pickle is a secure version [1]

Trusted Pickle - Python m

sourceforge.net/projects/trustedpickle/

sourceforge | Search | Browse | Enterprise | Blog | Help | Jobs | Log In or

SOLUTION CENTERS | Go Parallel | HTML5 | Smarter IT | Software Delivery | Performance | Data Management | Resources | Newsletters

Home / Browse / Cryptography / Trusted Pickle - Python module

Trusted Pickle - Python module Alpha

Brought to you by: gre7g

Summary | Files | Reviews | Support | Wiki | News | Discussion | Donate

Add a Review
2 Downloads (This Week)
Last Update: 2013-04-15

Download
TrustedPickle.zip

Browse All Files

Description

TrustedPickle is a Python module which lets you create and sign your data files. By using public/private key techniques, this module protects your users from loading malicious data files that others might claim you created. LEGAL FOR EXPORT.

Recommended Projects

Numerical Python
wxPython
wxWidgets

Latest Tech Jobs Powered by Dice

Summary

- Python has built-in support for text, binary, JSON, XML, and serialization files
- File handles should generally be used within `with` blocks
- The io module gives a file API to in-memory objects
- The os module enables cross-platform file operations