Storing & Exchanging Data with Python

A whirlwind tour of various approaches

It's All Bits

- Everything is just bits
  - In memory
  - On disk
  - Over network
- All data formats data are just agreed upon ways of interpreting bits
- Many, many ways of storing data
- Let's start simple

Raw File I/O

- Sometimes you just need to store something simple
  - Or something low level
  - Or custom
- Sometimes you just need a file

Reading & Writing (Binary) Files

Mode

```
f = open(“/path/to/file”, “rb”)
contents = f.read()    # contents is bytes
f.close()


f = open(“/path/to/file”, “wb”)  # create/truncate and then write
f.write(b”hello”)                # note, binary again
f.close()
```

Mode also allows for appending, simultaneous reading and writing etc

Aside: File objects as context managers

For quick operations on files:

```
with open(“/path/to/file”, “rb”) as f:
    … do stuff with f …
```

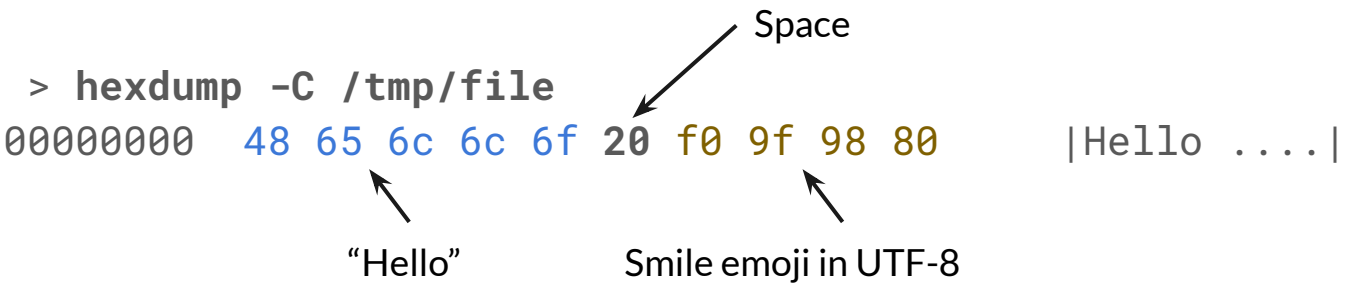
Avoids leaking file handles, even when things go wrong.

Reading & Writing Text Files

- When working with textual data, opening files in text mode is more natural
- Python handles encoding/decoding between strings and on-disk representation
- More convenient but less control
- Text files are the default: `open(..., "r")` is the same as `open(..., "rt")`
- Best to specify encoding or Python will pick a system default
    - If interoperability is important

Text File Encoding

```
>>> f = open("/tmp/file", "w",encoding="utf-8")
>>> f.write("Hello 😀")
>>> f.close()
```



More Text Files

```
>>> f = open("/tmp/file", "r",encoding="utf-8")
>>> print(f.read())
Hello 😀
```

- Text files can be read and written by line-by-line
- Newline handling via `newline` keyword arg
  - Universal newline handling
  - Python always gives you \n (LF / 0x0A)

File Buffering

- A file object may have a read or write buffer associated with them
- A buffer is just a chunk of memory holding data to be written
  - or read from disk but not consumed yet
- Improves performance by reducing the number of calls to operating system

Buffering Comparison

```
import time

def write_data(f):
      t0 = time.time()
      for _ in range(5*1024*1024):
          f.write(b'x')
      t1 = time.time()
      return t1 - t0

with open("out.dat", "wb", buffering=0) as f:
      unbuf_time  = write_data(f)

with open("out.dat", "wb", buffering=8192) as f:
      buf_time  = write_data(f)

print(f"unbuffered: {unbuf_time:.2}s")
print(f"buffered:   {buf_time:.2}s")
```

```
unbuffered: 4.5s
buffered:   0.38s
```

mmap - Memory-mapped file support

mmap maps a file to space on memory. It can also create an “anonymous” block of memory. In both cases, the mapped memory can be shared between threads and processes.

Has functionality that combines file objects and bytearrays. In addition, mmap objects have find, rfind, and move methods.

```
import mmap

with open("hello.txt", "wb") as f:
    f.write(b"Hello Python\n")

with open("hello.txt", "r+b") as f:
      with mmap.mmap(f.fileno(), 0) as mm:
            print(mm.readline())  # prints b"Hello Python!\n"
            print(mm[:5])  # prints b"Hello"
            mm.seek(5)
            mm.write(b" world!\n")
            mm.seek(0)
            print(mm.readline())  # prints b"Hello world!\n”
            mm.resize(17)
            mm.move(10, 6, len(b'world!\n'))
            mm[6:10] = b'big '
            print(mm[:])  # prints b'Hello big world!\n'
```

pickle

A Python-specific serialisation protocol. It can serialise most Python objects to a bytes-like object. It can even serialise functions and classes, but they must be importable via the script they are being de-serialised into.

Serialising Standard Library objects tend to be fine. Serialising more complicated 3rd party Python objects for reuse later can cause issues.

JSON

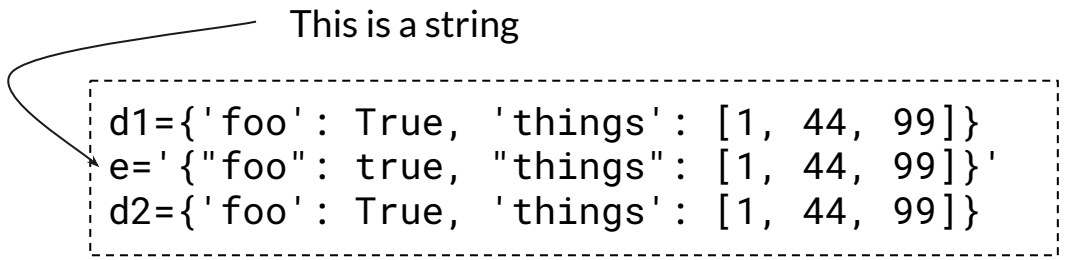
- The modern lingua franca of data exchange
- For better or worse has replaced XML
- Simple and somewhat limited
    - which explains it’s success
- Fairly human readable (when formatted)
- Great for nested, structured data
- Not size efficient
- Not particularly CPU efficient to parse

JSON in Python

- `json` package in standard library will suit most people's needs
- Mainly deals with built-in types (int, string, bool, float, dict etc)
- Handling for custom types can be added
- Main entry points:
  - `json.load()` / `.loads()` - parse JSON from a file/string
  - `json.dump()` / `.dumps()` - encode JSON to a file/string

JSON in Python Example

```
import json

d1 = {
     "foo": True,
     "things": [1, 44, 99],
}
print(f"{d1=}")

e = json.dumps(d1)
print(f"{e=}")

d2 = json.loads(e)
print(f"{d2=}")
```



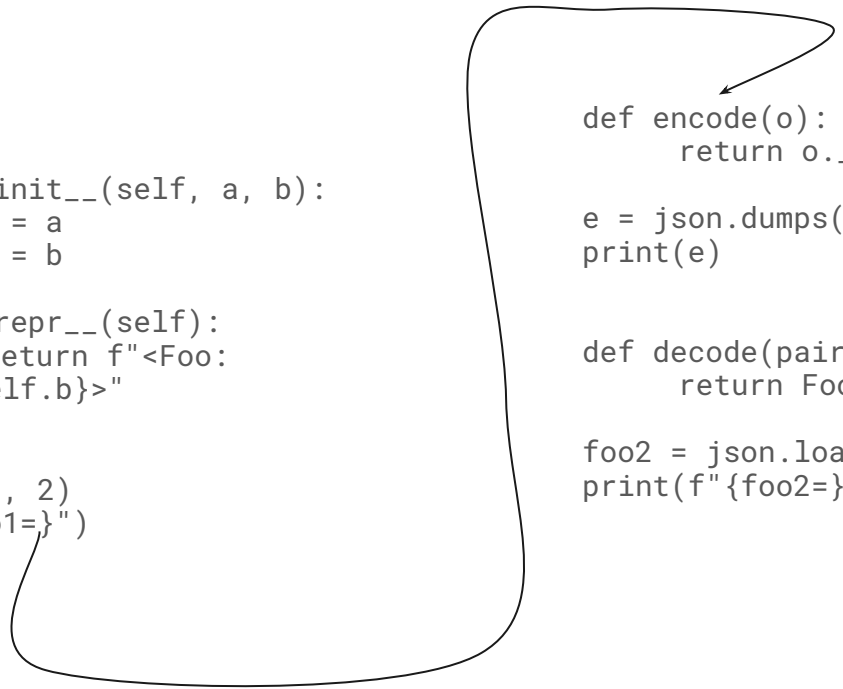
JSON In Python - Custom Types

- Encoded by with a custom `default` function
- Decoded with a custom `object_pairs_hook`
- Or use custom JSONEncoder and JSONDecoder
- Customising specific handling of ints and floats is also possible

JSON In Python - Custom Types Example

```
import json

class Foo:
     def __init__(self, a, b):
     self.a = a
     self.b = b

     def __repr__(self):
          return f"<Foo:
{self.a} {self.b}>"


foo1 = Foo(1, 2)
print(f"{foo1=}")
```

```
def encode(o):
     return o.__dict__

e = json.dumps(foo1, default=encode)
print(e)


def decode(pairs):
     return Foo(**dict(pairs))

foo2 = json.loads(e, object_pairs_hook=decode)
print(f"{foo2=}")
```



Custom JSON Encoding Output

```
foo=<Foo: 1 2>
{"a": 1, "b": 2}
foo2=<Foo: 1 2>
```

YAML, TOML and Friends

- There are many alternatives to JSON
- Similar yet different tradeoffs
- YAML
  - More human friendly ⇒ various gotcha
  - Much more complicated
- TOML
  - INI style
  - Few gotchas (stricter)
  - Human friendly
  - Nesting is unconventional

Binary Formats

- Faster to generate and parse
- More compact representation
- Type safety
- Not human readable
    - Tooling required
- DIY with `struct` package in standard library?
- Several options

MessagePack

- “It's like JSON, but fast and small”
- No schema
- Feels similar to use as the `json` package
- Good interoperability: *many* languages supported
- Primitive types + handling for custom type extensions
- https://msgpack.org/

MessagePack Example

```
import msgpack

x = {
     "foo": ["hello", 123],
     "bar": True,
}
print(f"{x=!r}")

e = msgpack.packb(x)   # Or dumps()
print(f"{e=!r}")

y = msgpack.unpackb(e) # Or loads()
print(f"{y=!r}")
```

```
x={'foo': ['hello', 123], 'bar': True}
e=b'\x82\xa3foo\x92\xa5hello{\xa3bar\xc3'
y={'foo': ['hello', 123], 'bar': True}
```

protobuf

- From Google
- Widely used
- Supported by many programming languages
- Core piece of gRPC
- Schema driven
- Good handling of field deprecation
- Code generation

protobuf Definitions

```
syntax = "proto2";

package contact;

message Contact {
  optional string name = 1;
  optional string email = 2;

  enum PhoneType {
      MOBILE = 0;
      HOME = 1;
      WORK = 2;
  }

  message PhoneNumber {
      optional string number = 1;
      optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 3;
}
```

protobuf - Compiling

```
protoc -I=. --python_out=. contact.proto
❯ ls
contact.proto  contact_pb2.py
```

protobuf - Writing

```
import contact_pb2

person = contact_pb2.Contact()
person.name = "Alice"
person.email = "alice@whoknows.net"
phone = person.phones.add()
phone.number = "123456789"
phone.type = contact_pb2.Contact.PhoneType.MOBILE
print(person)

encoded = person.SerializeToString()
print(encoded)
```

protobuf - Writing

```
name: "Alice"
email: "alice@whoknows.net"
phones {
  number: "123456789"
  type: MOBILE
}

b'\n\x05Alice\x12\x12alice@whoknows.net\x1a\r\n\t123456789\x10\x00'
```

protobuf - Reading

```
person2 = contact_pb2.Contact()
person2.ParseFromString(encoded)
print(person2)
```

Output is:

```
name: "Alice"
email: "alice@whoknows.net"
phones {
  number: "123456789"
  type: MOBILE
}
```

More on protobuf

Generated protobuf types aren't normal Python objects.

```
>>> import contact_pb2
>>> person = contact_pb2.Contact()

>>> person.name = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 99 has type int, but expected one of: bytes, unicode

>>> person.what = "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'what'
```

Tabular Data Formats

- csv/tsv
- Apache Arrow formats
- HDF5 via pytables

https://pandas.pydata.org/docs/user_guide/io.html

```
import numpy as np
import pandas as pd

index = pd.date_range("1/1/2000", periods=8)
index.name = 'date'
df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"])

df.to_csv('table_with_index.csv')
df.to_csv('table_wo_index.tsv', index=False, sep='\t')

df1 = pd.read_csv('table_with_index.csv', index_col='date')
df2 = pd.read_table('table_wo_index.tsv', sep='\t')

df.reset_index().to_feather('table.feather', compression='zstd', compression_level=1)
df.to_parquet('table.parquet')
df.to_hdf('table.h5', key='table')
```

Multi-dimensional data

- HDF5 via h5py and xarray
- netcdf4 via xarray

https://docs.xarray.dev/en/stable/user-guide/io.html

```
import numpy as np
import xarray as xr
import pandas as pd

ds = xr.Dataset(
      {"foo": (("x", "y"), np.random.rand(4, 5))},
      coords={
      "x": [10, 20, 30, 40],
      "y": pd.date_range("2000-01-01", periods=5),
      },
)

ds.to_netcdf('labelled_array.nc')

ds1 = xr.open_dataset('labelled_array.nc')
```

```
<xarray.Dataset>
Dimensions:  (x: 4, y: 5)
Coordinates:
  * x        (x) int64 10 20 30 40
  * y        (y) datetime64[ns] 2000-01-01
2000-01-02 ... 2000-01-04 2000-01-05
Data variables:
      foo    (x, y) float64 0.1355 0.8987 0.2956
0.9127 ... 0.1199 0.4378 0.2762
```

Embedded key-value DBs

- Oracle Berkeley DB
- Python dbm (dbm.gnu, dbm.dumb)
- Lightning Memory-Mapped Database (lmdb)
- sqlitedict

```
import shelve

key1 = 'foo'
data1 = ['big', 'list', 'of', 1, 2, 3]

d = shelve.open('persistent_dict.db', 'n')

d[key1] = data1
data = d[key1]
del d[key1]

flag = key1 in d
klist = list(d.keys())

d.close()
```