Understanding Python’s pandas source code

D. Gueorguiev 5/26/21

<https://github.com/pandas-dev/pandas>

Contents

[Preliminaries and third-party packages 1](#_Toc73295082)

[Preliminaries 1](#_Toc73295083)

[Google Flatbuffers 1](#_Toc73295084)

[Apache Arrow 5](#_Toc73295085)

[Swig 5](#_Toc73295086)

[NumPy 5](#_Toc73295087)

[Intel MKL 5](#_Toc73295088)

[OpenBLAS 5](#_Toc73295089)

[Atlas 5](#_Toc73295090)

# Preliminaries and third-party packages

## Preliminaries

The reader is assumed to have working knowledge of python v3, C and some C++.

## Google Flatbuffers

<https://google.github.io/flatbuffers/>

<https://github.com/google/flatbuffers>

From Flatbuffers Programmers Guide <https://google.github.io/flatbuffers/usergroup0.html>

Flatbuffers is efficient serialization library supporting various languages.

Benefits of using Flatbuffers:

* Access to serialized data without parsing/unpacking – what sets Flatbuffers apart is that it represents hierarchical data in a flat binary buffer in such way that it still can be accessed without parsing / unpacking while also still supporting data structure evolution (forward/backward compatibility)
* Memory efficiency and speed- the only memory needed to access your data. It requires 0 additional allocations in C++. Flatbuffers is also very suitable for use of mmap (or streaming) requiring only a part of the buffer to be in memory. Access is close to the speed of raw struct access with only one extra indirection (a kind of vtable) to allow for format evolution and optional fields. It is aimed at projects where spending time and space (many memory allocations) to be able to access or construct serialized data is undesirable such as in games or any other performance sensitive applications
* Flexible – Optional fields means not only do you get forwards and backwards compatibility (do not have to update all data with each new version). It also means you have a lot of choice in what data you write and what data you don’t and how you design data structures.
* Tiny code footprint – small amounts of generated code and just a single small header as the minimum dependency which is very easy to integrate.
* Strongly typed – Errors happen at compile time rather than manually having to write repetitive and error prone run-time checks. Useful code can be generated for you.
* Convenient to use – generated C++ code allows for terse access and construction code.
* Cross platform code with no dependencies

How to use Flatbuffers:

* Write a schema file that allows you to define the data structures you may want to serialize. Fields can have scalar types (int / floats of all sizes) or they can be : string, array of any type; reference to yet another object; or a set of possible objects (unions). Fields are optional and have defaults so they do not need to be present for every object instance.
* Use flatc (the FlatBuffer compiler) to generate C++ header (or Java/Kotlin/C#/Go/Python classes) with helper classes to access and construct the serialized data. This header (say mydata\_generated.h) only depends on flatbuffers.h which defines the core functionality.
* User the FlatBufferBuilder class to construct a flat binary buffer. The generated functions allow you to add objects to this buffer recursively often as simply as making a single function call.
* Store or send the buffer somewhere
* When reading it back you can obtain a pointer to the root object from the binary buffer and from there traverse it conveniently in place with object->field().

Small example:

Writing the Monster’s FlatBuffer Schema

To start working with FlatBuffers you first need to create a schema file which defines the format of each schema file you wish to serialize. Here is the schema that defines the template for our monsters:

// Example IDL file for our monster’s schema

namespace MyGame.Sample;

enum Color:byte { Red = 0, Green, Blue = 2}

union Equipment { Weapon } // optionally add more tables

struct Vec3 {

x:float;

y:float;

z:float;

}

table Monster {

pos:Vec3; // Struct

mana:short = 150;

hp:short = 100;

name:string;

friendly:bool = false (deprecated);

inventory:[ubyte]; // Vector of scalars

color:Color = Blue; // Enum

weapons:[Weapon]; // Vector of tables

equipped:Equipment; // Union

path:[Vec3]; // Vector of structs

}

table Weapon {

name:string;

damage:short;

}

root\_type Monster;

The schema starts with a namespace declaration. This determines the corresponding package/namespace for the generated code. In our example we have Sample namespace inside the MyGame namespace. Next we have an enum definition. In this example, we have enum of type byte, named Color. We have three values in this enum: Red , Green and Blue. We specify Red = 0 and Blue = 2, but we do not specify an explicit value for Green. Since the behavior of an enum is to increment if unspecified Green will receive the implicit value of 1. Following the enum is a union. The union in this example is not very useful as it only contains the one table (named Weapon). If we had created multiple tables that we would want the union to be able to reference we could add more elements to the union Equipment. After the union comes a struct Vec3, which represents a floating point vector with *3* dimensions. We use struct here instead of table as structs are ideal for modeling data structures which will not change since they use less memory and have faster lookup. The Monster table is the main object in our FlatBuffer. This will be used as a template to store our orc monster. We specify some default values for fields such as mana:short = 150. If unspecified, scalar fields like int, uint or float will be given a default of 0 while strings and tables will be given a default of null. Another thing to note is the line friendly: bool = false (deprecated); . Since you cannot delete fields from a table (to support backwards compatibility), you can set fields as deprecated, which will prevent the generation of accessors for this field in the generated code. The keyword deprecated can break legacy code that used that accessor. The Weapon table is a sub-table used within our FlatBuffer. It is used twice: once within the Monster table and once within the Equipment union. For our Monster it is used to populate a vector of tables via the weapons field within our Monster.

It is also the only table referenced by the Equipment union. The last part of the schema is the root\_type. The root type declares what will be the root table for the serialized data. In our case the root type is our Monster table. The scalar types can also use alias type names such as int16 instead of short and float32 instead of float. Thus we could also write the Weapon table as:

table Weapon {

name:string;

damage:int16;

}

More information about Flatbuffers Schema

Let us look again into the same example of Flatbuffers schema with small modifications:

// example IDL file

namespace MyGame;

attribute “priority”;

enum Color : byte { Red = 1; Green; Blue }

union Any { Monster, Weapon, Pickup }

struct Vec3 {

x:float;

y:float;

z:float;

}

table Monster {

pos:Vec3;

mana:short = 150;

hp:short= 100;

name:string;

friendly:bool = false (deprecated, priority: 1)

inventory:[ubyte];

color:Color = Blue;

test:Any;

}

root\_type Monster;

Weapon was defined earlier while Pickup is not defined in this example.

Tables

Tables are the main way of defining objects in FlatBuffers and consists of a name (here Monster) and a list of fields. Each field has a name, a type, and optionally a default value. If the default value is not specified in the schema, it will be 0 for scalar types, or null for other types. Some languages support setting scalar default to null. This makes the scalar optional.

Fields do not have to appear in the wire representation and you can choose to omit fields when constructing an object. You have the flexibility of adding fields without bloating your data. This design is also FlatBuffer’s mechanism for forward and backward compatibility. Note that:

* You can add new fields only at the end of a table definition. Older data will still read correctly and give you the default value when read. Older code will simply ignore the new field. If you want to have flexibility to use any order for the fields in your schema, you can manually assign ids (much like Protocol Buffers), see the id attribute below.
* You cannot delete fields you do not use anymore from the schema, but you can simply stop writing them into your data for almost the same effect. Additionally you can mark them as deprecated as in the example above which will prevent the generation of accessors in the generated C++, as a way to enforce the field not being used any more (this could break older code).
* You may change field names and table names but do not forget to rename the same in your code as well

Schema evolution examples – these are examples which clarify what happens when you change the schema. We have the following original schema:

table { a:int; b:int }

and we extend it:

table { a:int; b:int; c:int; }

This is OK. Code compiled with the old schema reading data generated with the new one will simply ignore the presence of the new field. Code compiled with the new schema reading old data will get the default value for c (which is 0 in this case since it is not specified).

table { a:int (deprecated); b:int; }

This is also ok. Code compiled with the old schema reading new data will now always get the default value for a since it is not present. Code compiled with the old schema reading newer data will now always get the default value for a since it is not present. Code compiled with the new schema now cannot read nor write a anymore (any existing code that tries to do so will result in compile errors), but can still read old data (they will ignore the field).

table { c:int; a:int; b:int; }

This is NOT ok, as this makes the schemas incompatible. Old code reading newer data will interpret c as if it was a, and new code reading old data accessing a will instead receive b.

table { c:int (id: 2); a:int (id: 0); b:int (id: 1) }

This is ok. If your intent was to order/group fields in a way that makes sense semantically, you can do so using explicit id assignment. Now we are compatible with the original schema and the fields can be ordered in any way, as long as it keeps the sequence of ids.

table { b:int; }

Not ok. We can remove a field only by deprecation regardless of wether we use explicit ids or not.

table { a:uint; b:uint; }

This is MAYBE ok, and only in the case where the type change is the same size, like here. If old data never contained any negative numbers this will be safe to do.

table { a:int = 1; b:int = 2; }

Generally NOT ok. Any older data written that had 0 values were not written to the buffer, and rely on the default value to be recreated. These will not have those values appear to 1 and 2 instead. There will be cases in which this is OK but care must be taken.

table { aa: int; bb: int }

Occasionally OK. You have renamed fields which will break all code and JSON files which use this schema, but as long as the change is obvious this is not incompatible with the actual binary since those ever address fields by id/offset.

Structs

Similar to a table only now none of the fields are optional and fields may not be added or deprecated. Structs may only contain scalars or other structs. Use this for simple objects where you are very sure no changes will ever be made (as quite clear in the example Vec3). Structs use less memory than tables and are even faster to access (they are always stored inline in their parent object and use no virtual table).

Types

Built-in scalar types are:

8 bit : byte (int8), ubyte (uint8), bool

16 bit:

## Apache Arrow

<https://github.com/apache/arrow>

<https://arrow.apache.org/docs/format/Columnar.html>

From <https://arrow.apache.org/overview/> :

Apache Arrow defines language-independent columnar memory format for flat and hierarchical data, organized for efficientanalytic operations on modern hardware like CPUs and GPUs. The Arrow memory format supports zero copy reads for fast data access without serialization overhead.

Apache Arrow is software development platform for building high-performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or one programming language to another. The defining component of Arrow is its in-memory columnar format which is a standardized, language-agnostic specification for representing structured, table-like datasets in memory. This format has rich data type system including nested and user-defined data types designed to support the needs for analytic database systems, data frame libraries, etc.

The Apache Arrow format allows computational routines and execution engines to maximize their efficiency when scanning and iterating large chunks of data. In particular, the contiguous columnar layout enables vectorization using the latest SIMD operations included in modern processors.

Assuming standard format for the data speeds up execution

Without standard format there are potentially a lot of conversion and transforming of data which is lots of unnecessary operations. Moving data from one system to another involves costly serialization and deserialization. In addition, common algorithms must be transformed / rewritten for each data format.

Arrow’s in-memory columnar format provides a solution to these kinds of problems allowing data transfers between between disparate systems to be achieved with very low cost. Additionally, the standardized format allows for reuse of libraries and algorithms.

The Arrow Columnar Format

The Arrow Columnar Format includes a language-agnostic in-memory data structure specification, metadata serialization, and a protocol for serialization and generic data transport. The new columnar format is created without the aid of existing implementation. For the implementation of the columnar format [Flatbuffers](https://github.com/google/flatbuffers) is used for metadata serialization purposes.

## Swig

<https://github.com/swig/swig>

<http://www.swig.org/tutorial.html>

## NumPy

<https://github.com/numpy/numpy>

<https://numpy.org/doc/stable/user/>

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices) and an assortment of routines for fast operations on arrays including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulations, etc.

At the core of the NumPy package is the ndarray object. This encapsulates n-dimensional arrays of homogenenous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have fixed size at creation, unlike Python lists which can grow dynamically. Changing the size of an ndarray will create a new array and delete the original
* The elements in a NumPy array are all required to be of the same data type , and thus will be the same size in memory. The exception is that one can have arrays of (Python including NumPy) objects thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than it is possible using Python built-in sequences
* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software just knowing how to use Python sequences is insufficient – one also need to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data is stored in two Python lists, **a** and **b**, we could iterate over each element as:

c = []

for i in range(len(a)):

c.append(a[i]\*b[i])

This produces the correct answer but if **a** and **b** each contain millions of numbers, we will pay the price for inefficiencies of looping in Python. We could achieve the same task much more quickly in C by writing (for clarity we neglect variable declarations, initializations, memory allocations):

for (i = 0; i < rows; i++) {

c[i] = a[i]\*b[i];

}

This saves all the overhead involved in interpreting Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of the data. In the case of 2-D array, for example, the C code abridged as before expands to:

for (i = 0; i < rows; i++) {

for (j = 0; j < columns; j++) {

c[i][j] = a[i][j]\*b[i][j];

}

}

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when ndarray is involved, but element-by-element operation is speedily executed by pre-compiled C code. In NumPy:

c = a \* b

does what the earlier examples do in near C speeds, but with code simplicity we expect from something based on Python. Indeed, the numpy idiom is even simpler. This last example illustrates two of NumPy’s features which are basis of its power – broadcasting and vectorization. Vectorization describes the absence of any explicit looping and indexing in the code. Looping and indexing are taking place behind the scenes in optimized pre-compiled C code. Vectorized code has many advantages among which are:

* Vectorized code is more concise and easier to read
* Fewer lines of code means fewer bugs
* The code closely resembles standard mathematical notation hence is more suitable for scientific computations

Broadcasting is the term used to describe the implicit element-by-element behavior of operations. In NumPy all operations not just arithmetic operations but logical, bit-wise, functional behave in this implicit element-by-element fashion i.e. they broadcast. Moreover, **a** and **b** could be multidimensional arrays of the same shape or a scalar and an array or even two arrays with different shapes provided that the smaller array is expandable to the shape of the larger such that the broadcast is unambiguous.

Data Types

Array Types and conversion between types

The primitive types supported are closely tied to those in C:

NumPy type C type Description

numpy.bool\_ bool Boolean (True or False) stored as a byte

numpy.byte signed char Platform defined

numpy.ubyte unsigned char Platform defined

numpy.short short Platform defined

numpy.ushort unsigned short Platform defined

numpy.intc int Platform defined

numpy.uintc unsigned int Platform defined

## Intel MKL

[https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html](https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html#gs.2pci5n)

## OpenBLAS

<https://www.openblas.net/>

<https://github.com/blas-lapack-rs/openblas-src>

## Atlas

<https://github.com/math-atlas/math-atlas>

<https://sourceforge.net/projects/math-atlas/>

<https://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software>