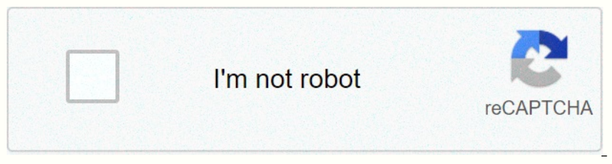

Continue

Numpy python binary format

Welcome to the plyfile Python module, which provides easy access to read and write ASCII files and PLY binaries. The PLY format is documented elsewhere. Installing python2 > 2.7 or python3 numpy > 1.8 Note: numpy 1.9 before version 1.9.2 has an error that breaks the byte exchange by manipulating the byte_order field of a PlyData instance. As a workaround, you can manually swap byte arrays using the .data.byteswap().newbyteorder() in addition to changing the byte_order. Optional dependencies setuptools (for installation via setup.py) tox (for the test suite) py.test and py (for the test suite) Installing the plyfile Quick form: pip install plyfile OR clone the repository and run from the project root: python setup.py install or simply copy plyfile.py into your GPL-compatible project. Running the Preferred test suite (more complete; requires tox and setuptools): tox -v Alternate (requires py.test and py): py.test test -v Usage Both deserialization and serialization of PLY file data are performed through PlyData and PlyElement instances. >>> from plyfile import PlyData, PlyElement For the following code examples, suppose the tet.ply file contains the following text: ascii bend format 1.0 comment only tetrahedron with colored faces vertex element 4 comment vertices of tetrahedron property float x float property and float z element float element face 4 property list uchar int vertex_indices property uchar red property uchar green property uchar blue end_header 0 0 0 1 1 0 1 1 1 1 0 3 0 1 2 255 255 255 3 0 2 3 255 0 0 3 0 1 3 0 255 0 3 1 2 3 0 0 255 (This file is available in the sample directory.) Reading a PLY file >>> plydata to PlyData.read('tet.ply') or >>> with open('tet.ply', 'rb') as f: ... plydata - PlyData.read(f) The static PlyData.read method returns an instance of PlyData, which is the plyfile representation of the data in a PLY file. An instance of PlyData has an Attribute elements, which is a list of PlyElement instances, each of which has a data attribute that is a numpy structured array that contains the numeric data. PLY file elements map to numpy structured arrays in a rather obvious way. For a list property on an item, the corresponding numpy field type is object, with members as numpy arrays (see example vertex_indices below). Specifically: >>> plydata.elements[0].name 'vertex' >>> plydata.elements[0].data[0] (0.0, 0.0, 0.0) >>> plydata.elements[0].data['x'] array([0., 1., 1.], dtype=float32) >>> array plydata['face'].data['vertex_indices'][0] 2], dtype-int32) elements and properties can be searched by name: >>> array plydata['vertex']['x'] ([0., 0., 1., 1.], dtype-float32) and elements can be indexed directly without explicitly passing through the data attribute : >>> plydata['vertex'][0] (0.0, 0.0, 0.0) The above expression is equivalent to plydata['vertex'].data[0]. PlyElement instances also metadata: >>> plydata.elements[0].properties (PlyProperty('x', 'float'), instances PlyProperty('y', 'float'), PlyProperty('z', 'float')) >>> plydata.elements[0].count 4 PlyProperty and PlyListProperty are used internally as a convenient intermediate representation of the properties of the PLY element that can be easily serialized into a PLY header (using str) or converted to numpy-compatible type descriptions (via the dtype method). It is not very common to manipulate them directly, But if necessary, you can access an item's property metadata as a tuple through the properties attribute (as illustrated above) or search by name: >>> plydata.elements[0].ply_property('x') PlyProperty('x', 'float') Many (but not necessarily all) types of incorrectly formatted input files will generate PlyParseError when PlyData is called. The string value of the PlyParseError instance (as well as the attribute element, row, and prop) provides additional context for the error if applicable. Creating a PLY File The first step is to get the data into numpy structured arrays. Note that there are some restrictions: Generally speaking, if you know the types of properties that a PLY file item can contain, you can easily infer constraints. For example, PLY files do not contain 64-bit integer or complex data, so they are not allowed. For convenience, non-scalar fields are allowed; will be serialized as list properties. For example, when constructing an expensive element, if all faces are triangles (a common occurrence), it is OK to have a vertex_indices field of type 'i4' and shape (3,) instead of type object and shape (). However, if the serialized PLY file is reread using plyfile, the vertex_indices property will be represented as an object-typed field, each of whose values is an array of type 'i4' and length 3. The reason is simply that the PLY format does not provide any way to find out that each vertex_indices field has length 3 without actually reading all the data, so plyfile has to assume that it is a variable-length property. However, see below (and examples/graphics.py) for an easy way to retrieve a two-dimensional array of a list property. For example, if we wanted to create the PLY vertex and face elements of the tet.ply data directly as numpy arrays for serialization purpose, we could do (as in test/test.py): >>> vertex á numpy.array([(0, 0, 0), ... (0, 1, 1), ... (1, 0, 1), ... (1, 1, 0)], ... dtype[('x', 'f4'), ('y', 'f4'), ... ('z', 'f4')]) >>> face of the letter files[(0, 1, 2], 255, 255, 255), ... ([0, 2, 3], 255, 0, 0), ... 1, 3], 0, 255, 0), ... ([1, 2, 3], 0, 0, 255)], ... dtype[('vertex_indices', 'i4', (3,)), ... ('red', 'u1'), ('green', 'u1'), ... ('blue', 'u1')]) Once you have properly structured the array, the static PlyElement.describe method can be used to create the required PlyElement instances: >>> the - PlyElement.describe(some_array, -PlyElement.describe(some_array, o >>> the - PlyElement.describe(some_array, 'some_name', ... comments ['comment1', ... 'comment2']) Note that you do not need to instantiate PlyProperty explicitly. All of this is done behind the scenes some_array.dtype.descr. A small setback here is that variable-length fields in a numpy array (that is, our representation of the properties in the PLY list) must have an object type, so the types of the list length and values in the serialized PLY file cannot be obtained only from the dtype attribute of the array. To simplify and predict simplicity, the default length value is 8-bit unsigned integer, and the default is 32-bit signed integer, which covers most use cases. Exceptions must be explicitly indicated: >>> the á plyElement.describe(some_array, 'some_name', ... val_dtypes "some_property': 'f8', ... len_dtypes"some_property': 'u4'-) You can now instantiate PlyData and serialize: >>> PlyData([the]).write('some_binary.ply') >>> PlyData([el], text-True).write('some_ascii.ply') - Force the byte order of the output to big-endian, regardless of the machine's native byte order >>> PlyData([el], ... byte_order">').write('some_big_endian_binary.ply') - Use a file object. Binary mode is used here, which will cause Unix-style line endings to be written to all systems. >>> with open('some_ascii.ply', mode 'wb') as f: ... PlyData([the], text-True).write(f) Multiple comments Comments are supported: >>> ply a PlyData([el], comments-['header comment']) >>> ply.comments ['header comment'] Starting with version 0.3, Comments obj_info are also supported: >>> ply á PlyData([el], obj_info-['obj_info1', 'obj_info2']) >>> ply.obj_info ['obj_info1' , 'obj_info2'] When typed, they will be placed after regular comments after the Line format. Comments can have leading white space, but trailing white space can be removed and should not be trusted. Comments cannot contain new embedded lines. Getting a two-dimensional array of a list property The PLY format provides no way to assert that all data in a given list property is of the same length, however, this is a relatively common occurrence. For example, all data that vertex_indices an expensive element will have a length three for a triangular mesh. In such cases, it is often much more convenient to have the data in a two-dimensional array, unlike a one-dimensional array of type object. This is a fairly easy way to get a two-dimensional matrix, we know the length of the row beforehand: >>> plydata a PlyData.read('tet.ply') >>> tri_data a plydata['face'].data['vertex_indices'] >>> triangles ? numpy.vstack(tri_data) Instance mutability A plausible code pattern is to read a PLY file on a PlyData instance, perform some operations on it, possibly modify the data and metadata instead, and write the result to a new file. This pattern is partially supported. Starting with version 0.4, the In-place mutations are supported: Modification of only matrix numeric data. Direct mapping to items in a PlyData instance. Change formatting by changing the text byte_order attributes of a PlyData instance. This will switch between ascii, binary_little_endian, and binary_big_endian PLY formats. Modify the comments and obj_info of a PlyData instance and modify the comments of a PlyElement instance. Mapping to an item's data. Note that this does not affect property metadata for properties, so for each property in the property list of the PlyElement instance, the data array must have a field with the same name (but possibly of a different type and possibly in a different order). The array can also have additional fields, but they will not be generated when you write the item to a PLY file. The output file properties appear as they are in the property list. If an array field has a different type than the corresponding PlyProperty instance, it will be converted when you type. Assign directly to an item's properties. Note that the data array is not touched and the previous note regarding the relationship between properties and data still applies: the field names of the data must be a subset of the property names in the properties, but they can be in a different order and specify different types. Change a val_dtype of the PlyProperty or PlyListProperty instance or the len_dtype of the PlyListProperty instance, which will perform the conversion on write. Modifying the name of an instance of PlyElement, PlyProperty, or PlyListProperty is not supported and an error will occur. To rename a property of a PlyElement instance, you can remove the property from the properties, Rename the data field and re-add the property to the properties with the new name by creating a new instance of PlyProperty or PlyListProperty: >>> from Plyfile import PlyProperty, The >>> property of the >>> face >>> face.properties >>> face.data.dtype.names a ['idx', 'r', 'g', 'b'] >>> face.properties á (PlyListProperty('idx', 'uchar', 'int'), ... PlyProperty('r', 'uchar'), ... PlyProperty('g', 'uchar'), ... PlyProperty('b', 'uchar')) Note that it is always safe to create a new instance of PlyElement or PlyData instead of modifying one instead, and this is the recommended style: >>> - Recommended: >>> plydata -PlyData([plydata['face'], plydata['vertex']], text-False, byte_order"<') >>> 'Also supported: >>> plydata.elements ' [plydata['face'], plydata['vertex']] >>> plydata.text ? False >>> ? '<' >>> plydata.comments ? [] >>> plydata.obj_info <3> <7> [] Objects created by this library do not claim ownership of the other objects they reference, which has implications for both styles (create new instances and modify instead). For example, multiple instances of PlyData can contain a single instance of PlyElement, but modifying that instance will affect all containing instances of PlyData. Frequently Asked Questions >>>: Here is a two-dimensional array containing vertex indexes. >>> face_data numpy.array([[0, 1, 2], [3, 4, 5]], dtype'i4') >>> - PlyElement.describe requires a one-dimensional structured array. >>> ply_faces numpy.empty(len(faces), ... dtype-['vertex_indices', 'i4', (3,),)]) >>> ply_faces['vertex_indices'] - face_data face >>> - PlyElement.describe(ply_faces, 'face') In Python 3, you will probably run into problems because sys.stdout is a text mode sequence and plyfile generates binary data, even for ASCII format PLY files: >>> import sys >>> plydata.write(sys.stdout) Traceback (latest call): File <stdin>, line 1, in <module>the .../python-file plyfile/plyfile.py, line 411, in write stream.write(self.header.encode('ascii')) argument TypeError: write() must be str, no bytes There are some ways to avoid this. Write to a named file instead. On Linux and some other Unix-likes, you can access stdout through the file named /dev/stdout: >>> plydata.write('/dev/stdout') Use sys.stdout.buffer: >>> plydata.write(sys.stdout.buffer) (source: Design philosophy and reasoning The plyfile design philosophy can be summarized as follows. Be familiar with numpy users and reuse existing idioms and concepts when possible. Favor simplicity over power or ease of use. Supports all valid PLY files. Familiarity For the most part, PLY concepts map very well to Python and specifically numpy, and the leverage that has strongly influenced the design of this package. The elements attribute of a PlyData instance is simply a list of PlyElement instances, and the data attribute of a PlyElement instance is a numpy array, and a list property field of a PLY element datum is referenced in the data attribute by an object type with the value of another numpy array, and so on. Simplicity When applicable, we prefer simplicity over power or ease of use. Therefore, list property types in PlyElement.describe always have the same default value, rather than, for example, that is obtained by looking at an array element. (What element? What happens if the array has zero length? Whatever default value we could choose in that case could lead to subtle case errors if the user is not vigilant.) In addition, all input and output is done in a socket: all arrays must be created in advance instead of streaming. Generality and issues of interpretation Our goal is to give to all valid PLY files. However, exactly what constitutes a valid file is not obvious, as there does not appear to be a single complete and consistent description of the PLY format. Even Greg Turk's authoritarian .txt has some problems. Comment Location Where can comments appear in the header? It appears that in all official examples, all comments immediately follow the format line, but the document language ni</module> </stdin> </stdin> nor does it explicitly allow comments to be placed elsewhere. Therefore, it is unclear whether comments can appear anywhere in the header or should immediately follow the line format. At least one popular PLY file reader chokes on comments before the format line. plyfile accepts comments anywhere in the header in the input, but only places them in a few limited places in the output, that is, immediately after the format and element lines. Element names and properties Another ambiguity is names: which strings are allowed as element names and PLY properties? plyfile accepts as input any name that does not contain spaces, but this is certainly too generous. This may not be as important, however: although the names are theoretically arbitrary, in practice, most of PLY's element and property names probably come from a small finite set (face, x, nx, green, etc.). Property syntax A more serious problem is that the PLY format specification appears to be inconsistent with property definition syntax. In some examples, it uses the Syntax property type, and in others, the Name property supports only the first, which appears to be de facto standard. Header line terminations The specification explicitly indicates that header lines must end with carriage returns, but this rule does not appear to be followed by anyone, including the C-language PLY implementation of Greg Turk, the author of the format. Here, we follow the common practice and output Unix-style line endings (no carriage returns) but accept any line termination style in the input files. More examples Examples beyond the scope of this document and testing are in the examples directory. Page 2 View 17 stars 285 Fork 55 You can't perform that action right now. You are logged in with another tab or window. Reload to refresh the session. You are logged out of another tab or window. Reload to refresh the session. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can create better products. Learn more. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can create better products. You can always update your selection by clicking Cookie Preferences at the bottom of the page. For more information, please see our Privacy Statement. We use cookies to perform essential website functions, for example, are used to log in. More information Always On We use analytics cookies to understand how you use our websites so that we can improve them, for example, they are used to collect information about the pages you visit and how many clicks you need to perform a task. Learn more