

# <sup>1</sup> pyfive: A pure-python HDF5 reader

<sup>2</sup> **Bryan Lawrence**  <sup>1</sup>, **Ezequiel Cimadevilla**  <sup>2</sup>, **Wout De Nolf**  <sup>6</sup>, **David Hassell**  <sup>1</sup>, **Jonathan Helmus**<sup>3</sup>, **Brian Maranville**  <sup>4</sup>, **Kai Mühlbauer**  <sup>5</sup>, and <sup>4</sup> **Valeriu Predoi** 

<sup>5</sup> 1 NCAS-CMS, Meteorology Department, University of Reading, Reading, United Kingdom.  <sup>2</sup>  
<sup>6</sup> Instituto de Física de Cantabria (IFCA), CSIC-Universidad de Cantabria, Santander, Spain. <sup>3</sup> TBD <sup>4</sup>  
<sup>7</sup> NIST Center for Neutron Research <sup>5</sup> Institute of Geosciences, Meteorology Section, University of Bonn,  
<sup>8</sup> Germany. <sup>1</sup> European Synchrotron Radiation Facility (ESRF), Grenoble, France.

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

---

Editor: [Open Journals](#) 

Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

## <sup>9</sup> Summary

<sup>10</sup> pyfive (<https://pyfive.readthedocs.io/en/latest/>) is an open-source thread-safe pure Python package for reading data stored in HDF5. While it is not a complete implementation of all the specifications and capabilities of HDF5, it includes all the core functionality necessary to read gridded datasets, whether stored contiguously or with chunks, and to carry out the necessary decompression for the standard options. All data access is fully lazy, the data is only read from storage when the numpy data arrays are manipulated. Originally developed some years ago, the package has recently been upgraded to support lazy access, and to add missing features necessary for handling all the environmental data known to the authors. It is now a realistic option for production data access in environmental science and more widely. The API is based on that of h5py (which is a python shimmy over the HDF5 c-library, and hence is not thread-safe), with some API extensions to help optimise remote access. With these extensions, coupled with thread safety, many of the limitations precluding the efficient use of HDF5 (and NetCDF4) on cloud storage have been removed.

## <sup>23</sup> Statement of need

<sup>24</sup> HDF5<sup>1</sup> ([Folk et al., 2011](#)) is probably the most important data format in physical science, used across the piste. It is particularly important in environmental science, particularly given the fact that NetCDF4<sup>2</sup> ([Rew et al., 2006](#)) is HDF5 under the hood. From satellite missions, to climate models and radar systems, the default binary format has been HDF5 for decades. While newer formats are starting to get mindshare, there are petabytes, if not exabytes, of existing HDF5, and there are still many good use-cases for creating new data in HDF5. However, despite the history, there are few libraries for reading HDF5 file data that do not depend on the official HDF5 library maintained by the HDFGroup, and in particular, apart from pyfive, in Python there are none that cover the needs of environmental science. While the HDF5 c-library is reliable and performant, and battle-tested over decades, there are some caveats to depending upon it: Firstly, it is not thread-safe, secondly, the code is large and complex, and should anything happen to the financial stability of The HDF5group, it is not obvious the C-code could be maintained. Finally, the code complexity also meant that it is not suitable for developing bespoke code for data recovery in the case of partially corrupt data. From a long-term curation perspective both of these last two constraints are a concern.

<sup>39</sup> The original implementation of pyfive (by JH), which included all the low-level functionality to deal with the internals of an HDF5 file was developed with POSIX access in mind. The

<sup>1</sup><https://www.hdfgroup.org/solutions/hdf5/>

<sup>2</sup><https://www.unidata.ucar.edu/software/netcdf>

41 recent upgrades were developed with the use-case of performant remote access to curated  
42 data as the primary motivation, but with additional motivations of having a lightweight HDF5  
43 reader capable of deploying in resource or operating-system constrained environments (such  
44 as mobile), and one that could be maintained long-term as a reference reader for curation  
45 purposes. The lightweight deployment consequences of a pure-python HDF5 reader need no  
46 further introduction, but as additional motivation we now expand on the issues around remote  
47 access and curation.

48 Thread safety has become a concern given the wide use of Dask<sup>3</sup> in python based analysis  
49 workflows, and this, coupled with a lack of user knowledge about how to efficiently use HDF5,  
50 has led to a community perception that HDF5 is not fit for remote access (especially on cloud  
51 storage). Issues with thread safety arise from the underlying HDF5 c-library, and cannot be  
52 resolved in any solution depending on that library, hence the desire for a pure python solution.  
53 Remote access has been bedevilled by the widespread need to access remotely data which has  
54 been chunked and compressed, combined with the use of HDF5 data which was left in the  
55 state it was when the data was produced - often with default unsuitable chunking (Rew, 2013)  
56 and with interleaved chunk indexes and data. Solutions have mainly consisted of reformatting  
57 the data (and rechunking it at the same time) or utilising kerchunk mediated direct access  
58 to chunked HDF5 data<sup>4</sup>. However, in practice using kerchunk requires the data provider to  
59 generate kerchunk indices to support remote users, and it leads to issues of synchronicity  
60 between indices and changing datasets.

61 This version of pyfive was developed with these use-cases in mind. There is now full support  
62 for lazy loading of chunked data, and methods are provided to give users all the benefits of  
63 using kerchunk, but without the need for a priori generation. Because pyfive can access and  
64 cache (in the client) the b-tree (index) on a variable-by-variable basis, most of the benefits of  
65 kerchunk are gained without any of the constraints. However, the kerchunk index is always a  
66 contiguous object accessible with one get transaction, this is not necessarily the case with the  
67 b-tree, unless the source data has been repacked to ensure contiguous metadata using a tool  
68 like h5repack. Much of the community is unaware of the possibility of repacking the index  
69 metadata, and this together with relatively opaque information about the internal structure  
70 of files (and hence the necessity or otherwise of such repacking), means that repacking is  
71 rarely done. To help with this process, pyfive also includes extensions to expose information  
72 about how data and indexes are distributed in the files. With these tools, index extraction  
73 with pyfive can be comparable in performance to obtaining a kerchunk index, and completely  
74 opaque to the user.

75 With the use of pyfive, suitably repacked and rechunked HDF5 data can now be considered  
76 “cloud-optimised”, insofar as with lazy loading, improved index handling, and thread-safety,  
77 there are no “format-induced” constraints on performance during remote access. To aid in  
78 discovering whether or not a given HDF5 dataset is cloud-optimised, pyfive also now provides  
79 simple methods to expose information about file layout - both in API extensions, and in a  
80 new p5dump utility packaged with the pyfive library, which provides (in the default view)  
81 functionality similar to ncdump, and when used with p5dump -s, information about storage  
82 characteristics.

83 The issues of the dependency on a complex code maintained by one private company in the  
84 context of maintaining data access (over decades, and potentially centuries), can only be  
85 mitigated by ensuring that the data format is well documented, that data writers use only the  
86 documented features, and that public code exists which can be relatively easily maintained.  
87 The HDF5group have provided good documentation for the core features of HDF5 which  
88 include all those of interest to the weather and climate community who motivated this reboot  
89 of pyfive, and while there is a community of developers beyond the HDF5 group (including  
90 some at the publicly funded Unidata institution), recent events suggest that given most of

<sup>3</sup><https://www.dask.org/>

<sup>4</sup><https://fsspec.github.io/kerchunk/>

91 those developers and their existing funding are US based, some spreading of risk would be  
 92 desirable. To that end, a pure python code, which is relatively small and maintained by an  
 93 international constituency, alongside the existing c-code, provides some assurance that the  
 94 community can maintain HDF5 access for the foreseeable future. A pure python code also  
 95 makes it easier to develop scripts which can work around data and metadata damage should  
 96 they occur.

## 97 Examples

98 We now introduce three aspects of the new functionality that pyfive now provides: remote  
 99 access, configurable lazy loading, and determining whether files are cloud optimised.

## 100 Remote Access

101 A notable feature of the recent pyfive upgrade is that it was carried out with thread-safety  
 102 and remote access using fsspec (file-system-spec.readthedocs.io) in mind. We provide two  
 103 examples of using pyfive to access remote data, one in S3, and one behind a modern http  
 104 web server:

105 For accessing the data on S3 storage, we will have to set up an s3fs virtual file system, then  
 106 pass it to Pyfive:

```
import pyfive
import s3fs
# storage options for an anon S3 bucket
# there are also caching options for the s3 middleware, not shown here
storage_options = {
    'anon': True,
    'client_kwargs': {'endpoint_url': "https://s3server.ac.uk"}
}
fs = s3fs.S3FileSystem(**storage_options)
file_uri = "s3-bucket/myfile.nc"
with fs.open(file_uri, 'rb') as s3_file:
    nc = pyfive.File(s3_file)
    dataset = nc[var]
```

107 for an HTTPS data server, the usage is similar:

```
import fsspec
import pyfive
# there are also caching options for the fsspec middleware, not shown here
fs = fsspec.filesystem('http')
with fs.open("https://site.com/myfile.nc", 'rb') as http_file:
    nc = pyfive.File(http_file)
    dataset = nc[var]
```

108 This is of course exactly the same pattern as remote access using h5py, and that is by design -  
 109 to make moving to pyfive easy for users!

## 110 Lazy Loading

111 A key tenet of efficient remote access is that variable inspection is quick and involves the  
 112 minimum of network traffic between storage and the client application.  
 113 However when this is coupled with the common pattern of using Dask, some flexibility in what  
 114 is loaded when is beneficial.

115 By default when one inspects the contents of a file using pyfive v1.0 nothing more is read  
 116 from the file than the names of the variables ("datasets" in the language of HDF5): for

117 example:

```
118     with pyfive.File('myfile.h5','r') as f:
119         variables_in_file = [v for v in f]
```

118 involves nothing more than getting a set of variable names. When one wishes to inspect these  
 119 variables:

```
120     with pyfive.File('myfile.h5','r') as f:
121         temp = f['temp']
122         print(temp[1:10])
```

120 the default in pyfive v1.0 is to get only the metadata associated with each variable - but  
 121 crucially at this point the b-tree index is also loaded and the variable can now be accessed  
 122 outside the context manager. Data loading is now completely lazy and the variable instance  
 123 (`temp`) has all the information needed to extract data as needed. This is done so that in Dask  
 124 applications, when one passes each Dask computational chunk a portion of the variable, each  
 125 such Dask chunk has already got the index, and when it does want to load data, it can be as  
 126 efficient as possible.

127 However, there are situations where loading the b-tree at variable instantiation is not wanted,  
 128 and all is wanted is to be able to view all the variable attributes. To support this option pyfive  
 129 also offers the `get_lazy_view` file method, so one can do:

```
130     with pyfive.File('myfile.h5','r') as f:
131         temp = f.get_lazy_view('temp')
132         print(temp.attrs())
```

130 It is still possible to access the data and the b-tree index is loaded when data access is first  
 131 attempted.

132 This extra lazy view is new functionality. It would obviously have been possible to make the  
 133 default not load the b-tree, but in the opinion of the current pyfive maintainers loading the  
 134 b-tree at variable instantiation is likely more consistent with user expectations as it is more  
 135 equivalent to the behaviour of other packages like h5py.

## 136 Cloud Optimisation

137 To be fully cloud optimised - as defined by Stern et al. (2022) - an HDF5 file needs to have  
 138 a contiguous index for each variable, and the chunks for each variable need to be sensibly  
 139 chosen and broadly contiguous within the file. When these criteria are met, indexes can be  
 140 read efficiently, and middleware such as fsspec can make sensible use of readahead caching  
 141 strategies.

142 HDF5 data files direct from simulations and instruments are often not in this state as information  
 143 about the number of variables, the number of chunks per variable, and the compressed size of  
 144 those variables is not known as the data is being produced.

145 In such cases the data is also often not chunked along the dimensions being added to as the  
 146 file is written (since it would have to be buffered first).

147 Of course, once the file is produced, such information is available. Metadata can be repacked to  
 148 the front of the file and variables can be rechunked and made continuous - which is effectively  
 149 the same process undertaken when HDF5 data is reformatted to other cloud optimised formats.

150 The HDF5 library provides a tool "h5repack" which can do this, provided it is driven with  
 151 suitable information about required chunk shape and the expected size of metadata fields.  
 152 Versions (>1.0) of pyfive supports both a method to query whether such repacking is necessary,  
 153 and to extract necessary parameters.

154 In the following example we compare and contrast the unpacked and repacked version of a  
 155 particularly pathological file, and in doing so showcase some of the pyfive API extensions

156 which help us understand why it is pathological, and how to address those issues for repacking.

157 If we extract just a piece of the output of p5dump -s on this file (which has surface wind  
158 velocity at three hour intervals for one hundred years):

```

159 float64 time(time) ;
160     time:_standard_name = "time" ;
161     time:_n_chunks = 292192 ;
162     time:_chunk_shape = (1,) ;
163     time:_btree_range = (31808, 19854095942) ;
164     time:_first_chunk = 9094 ;
165
166 float32 uas(time, lat, lon) ;
167     uas:_Storage = "Chunked" ;
168     uas:_n_chunks = 292192 ;
169     uas:_chunk_shape = (1, 143, 144) ;
170     uas:_btree_range = (28672, 19854809382) ;
171     uas:_first_chunk = 36520 ;

```

172 we can immediately see that this will be a problematic file! The b-tree index is clearly interleaved  
173 with the data (compare the first chunk address with last index addresses of the two variables),  
174 and with a chunk dimension of (1,), any effort to use the time-dimension to locate data of  
175 interest will involve a ludicrous number of 1 number reads (all underlying libraries read the data  
176 one chunk at a time). It would feel like waiting for the heat death of the universe if one was  
177 to attempt to manipulate this data stored on an object store!

178 It is relatively easy (albeit slow) to use h5repack to fix this - e.g see Hassell & Cimadevilla  
179 Alvarez (2025) - after which we see:

```

180 float64 time(time) ;
181     time:_Storage = "Chunked" ;
182     time:_n_chunks = 1 ;
183     time:_chunk_shape = (292192,) ;
184     time:_btree_range = (11861, 11861) ;
185     time:_first_chunk = 40989128 ;
186     time:_compression = "gzip(4)" ;
187 float32 uas(time, lat, lon) ;
188     uas:_Storage = "Chunked" ;
189     uas:_n_chunks = 5844 ;
190     uas:_chunk_shape = (50, 143, 144) ;
191     uas:_btree_range = (18663, 347943) ;
192     uas:_first_chunk = 41041196 ;
193     uas:_compression = "gzip(4)" ;

```

194 Now data follows indexes, the time dimension is one chunk, and there is a more sensible  
195 number of actual data chunks. While this file would probably benefit from splitting, with a  
196 contiguous set of indexes, it is now possible to exploit this data via S3.

197 All the metadata shown in this dump output arises from pyfive extensions to the  
198 pyfive.h5t.DatasetID class. pyfive also provides a simple flag: consolidated\_metadata  
199 for a File instance, which can take values of True or False for any given file, which simplifies  
200 at least the “is the index packed at the front of the file?” part of the optimisation question -  
201 though inspection of chunking is a key part of the workflow necessary to determine whether or  
202 not a file really is optimised for cloud usage.

## 203 Acknowledgements

204 Most of the developments outlined here that have occurred since V0.5 (primarily authored  
205 by JH) have been supported by the UK Met Office and UKRI via 1) UK Excalibur Exascale  
206 programme (project ExcaliWork), 2) the UKRI Digital Research Infrastructure programme  
207 (project WacaSoft), and 3) the national capability funding of the UK National Center for  
208 Atmospheric Science (NCAS). Ongoing maintenance of pyfive is expected to continue under  
209 the auspices of that NCAS national capability funding.

## 210 References

- 211 Folk, M., Heber, G., Koziol, Q., Pourmal, E., & Robinson, D. (2011). An overview of the HDF5  
212 technology suite and its applications. *Proceedings of the EDBT/ICDT 2011 Workshop on*  
213 *Array Databases*, 36–47. <https://doi.org/10.1145/1966895.1966900>
- 214 Hassell, D., & Cimadevilla Alvarez, E. (2025). *Cmip7repack: Repack CMIP7 netCDF-4*  
215 *datasets*. Zenodo. <https://doi.org/10.5281/zenodo.17550920>
- 216 Rew, R. (2013). Chunking data: Choosing shapes. In *News @ Unidata*. <https://www.unidata.ucar.edu/blo>  
217 *veloper/en//entry/chunking\_data\_choosing\_shapes*.
- 218 Rew, R., Hartnett, E., & Caron, J. (2006). NetCDF-4: Software implementing an enhanced  
219 data model for the geosciences. *22nd International Conference on Interactive Information*  
220 *Processing Systems for Meteorology, Oceanography and Hydrology*.
- 221 Stern, C., Abernathey, R., Hamman, J., Wegener, R., Lepore, C., Harkins, S., & Merose, A.  
222 (2022). Pangeo forge: Crowdsourcing analysis-ready, cloud optimized data production.  
223 *Frontiers in Climate*, 3. <https://doi.org/10.3389/fclim.2021.782909>