hictkpy Documentation

Roberto Rossini

Dec 03, 2025

CONTENTS

hictkpy provides Python bindings to hictk, a blazing fast toolkit to work with .hic and .cool files.

CHAPTER

ONE

DOCUMENTATION FORMATS

You are reading the PDF version of the documentation.

The live HTML version of the documentation is available at https://hictkpy.readthedocs.io/en/stable/.

Installation

hictkpy can be installed using pip or conda with e.g., `pip install 'hictkpy[all]'`. Refer to *Installation* for more details.

How to cite this project?

Please use the following BibTeX template to cite hictkpy in scientific discourse:

```
@article{hictk,
    author = {Rossini, Roberto and Paulsen, Jonas},
    title = "{hictk: blazing fast toolkit to work with .hic and .cool files}",
    journal = {Bioinformatics},
    volume = {40},
    number = {7},
    pages = {btae408},
    year = {2024},
    month = {06},
    issn = {1367-4811},
    doi = {10.1093/bioinformatics/btae408},
    url = {https://doi.org/10.1093/bioinformatics/btae408},
    eprint = {https://academic.oup.com/bioinformatics/article-pdf/40/7/btae408/
↪58385157/btae408.pdf},
}
```

CHAPTER

TWO

INSTALLATION

hictkpy can be installed in various ways.

2.1 PIP

```
pip install 'hictkpy[all]'
```

This will install hictkpy together with all its third-party dependencies.

It is also possible to install hictkpy with a minimal set of dependencies with one of the following commands:

```
pip install hictkpy  # this target has no runtime dependencies!
pip install 'hictkpy[numpy]'
pip install 'hictkpy[pandas]'
pip install 'hictkpy[pyarrow]'
pip install 'hictkpy[scipy]'
```

In general, Pandas and PyArrow are required when hictkpy is returning data using `pandas.DataFrame` or `pyarrow.Table`, such as when calling *hictkpy.PixelSelector.to_pandas()* or *hictkpy.BinTable.to_df()*.

NumPy is required when calling methods returning data as `numpy.array`, such as *hictkpy.PixelSelector.to_numpy()* or certain overloads of *hictkpy.BinTable.get_ids()*.

SciPy is required when fetching interactions as sparse matrix, such as *hictkpy.PixelSelector.to_coo()* and *hictkpy.PixelSelector.to_csr()*

In case applications call methods depending on missing third-party dependencies, hictkpy will raise an exception like the following:

```
In [3] f.fetch().to_numpy()

ModuleNotFoundError: No module named 'numpy'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: To enable numpy support, please install numpy with: pip install
↪'hictkpy[numpy]'
Alternatively, you can install hictkpy with all its dependencies by running: pip␣
↪install 'hictkpy[all]'
```

2.2 Conda (bioconda)

```
conda install -c conda-forge -c bioconda hictkpy
```

2.3 From source

Building hictkpy from source should not be necessary for regular users, as we publish pre-built wheels for Linux, macOS, and Windows for all Python versions we support (currently these include all CPython versions from 3.10 up until 3.14). For a complete and up-to-date list of available wheels refer to the Download files page on hictkpy's homepage on PyPI.

Building hictkpy's wheels from source requires a compiler toolchain supporting C++17, such as:

- GCC 8+
- Clang 8+
- Apple-Clang 10.0+
- MSVC 19.12+

Based on our testing, hictkpy's wheels compiled on Linux using Clang are noticeably faster than those compiled with GCC. For this reason we recommend building hictkpy using a modern version of Clang whenever possible. This can be achieved by redefining the `CC` and `CXX` environment variables before running pip (e.g. `CC=clang CXX=clang++ pip install ...`).

Furthermore, the following tools are required:

- git 2.7+
- make or ninja

2.3.1 Installing the latest version from the main branch

```
pip install 'hictkpy[all] @ git+https://github.com/paulsengroup/hictkpy.git@main'
```

2.3.2 Installing version corresponding to a git tag

```
pip install 'hictkpy[all] @ git+https://github.com/paulsengroup/hictkpy.git@v1.4.0'
```

2.3.3 Installing from a release archive

```
pip install 'hictkpy[all] @ https://pypi.python.org/packages/source/h/hictkpy/hictkpy-
↪1.4.0.tar.gz'
```

2.3.4 Running the automated tests

When building hictkpy from source we highly recommend running the automated test suite before using hictkpy in production.

This can be achieved in several ways. Here is an example:

```
git clone https://github.com/paulsengroup/hictkpy.git

cd hictkpy

# make sure to run tests for the same version/tag/commit used to build hictkpy
git checkout v1.4.0

```

(continues on next page)

(continued from previous page)

```
# if you installed hictkpy in a venv make sure to install pytest in the venv
pip install pytest

pytest test/
```

All tests are expected to pass. Do not ignore test failures!

However, it is expected that some test cases will be skipped (especially if not all optional dependencies were installed)

2.3.5 Notes

Building hictkpy requires several dependencies that are not needed after the build process. Some of these dependencies are installed using Conan, which creates several files under `~/.conan2`. If you don't need Conan for other purposes feel free to delete the `~/.conan2` once the build process completes successfully.

If you do not want to use Conan for dependency management you can set the `HICTKPY_PROJECT_TOP_LEVEL_INCLUDES` environment variable to an empty string. See section `[tool.scikit-build.cmake.define]` in the pyproject.toml file for the list of CMake variables that can be overridden by defining the appropriate environment variables.

CHAPTER

THREE

QUICKSTART

hictkpy provides Python bindings for hictk through nanobind.

hictkpy.File can open .cool and .hic files and can be used to fetch interactions as well as file metadata.

The examples in this section use the file 4DNFIOTPSS3L.hic, which can be downloaded from the 4D Nucleome Data Portal here.

3.1 Opening files

```
In [1]: import hictkpy as htk

# .mcool and .cool files are also supported
In [2]: f = htk.File("4DNFIOTPSS3L.hic", 10_000)

In [3]: f.path()
Out[3]: '4DNFIOTPSS3L.hic'
```

Important

The above example assigns the *hictkpy.File* directly to variable `f` for simplicity. Always prefer using context managers (e.g., the `with` keyword) when opening files using *hictkpy*:

```
with htk.File("4DNFIOTPSS3L.hic", 10_000) as f:
  # use the file
```

3.2 Reading file metadata

```
In [4]: f.resolution()
Out[4]: 10000

In [5]: f.chromosomes()
Out[5]:
{'2L': 23513712,
 '2R': 25286936,
 '3L': 28110227,
 '3R': 32079331,
 '4': 1348131,
 'X': 23542271,
 'Y': 3667352}

In [6]: f.attributes()
Out[6]:
```

(continues on next page)

(continued from previous page)

```
{'bin_size': 10000,
 'format': 'HIC',
 'format_version': 8,
 'assembly': '/var/lib/cwl/stgb25a903a-ebb6-4a56-bf3f-90bd84a40bf4/4DNFIBEEN92C.chrom.
↪sizes',
 'format-url': 'https://github.com/aidenlab/hic-format',
 'nbins': 13758,
 'nchroms': 7}
```

3.3 Fetch interactions

Interactions can be fetched by calling the *hictkpy.File.fetch()* method on *hictkpy.File* objects.

hictkpy.File.fetch() returns *hictkpy.PixelSelector* objects, which are very cheap to create.

```
# Fetch all interactions (genome-wide query) in COO format (row, column, count)
In [7]: sel = f.fetch()

# Fetch all interactions (genome-wide query) in bedgraph2 format
In [8]: sel = f.fetch(join=True)

# Fetch KR-normalized interactions
In [9]: sel = f.fetch(normalization="KR")

# Fetch interactions for a region of interest
In [10]: sel = f.fetch("2L:10,000,000-20,000,000")

In [11]: sel = f.fetch("2L:10,000,000-20,000,000", "X")

In [12]: sel.nnz()
Out[12]: 2247057

In [13]: sel.sum()
Out[13]: 7163361
```

3.3.1 Fetching interactions as pandas DataFrames

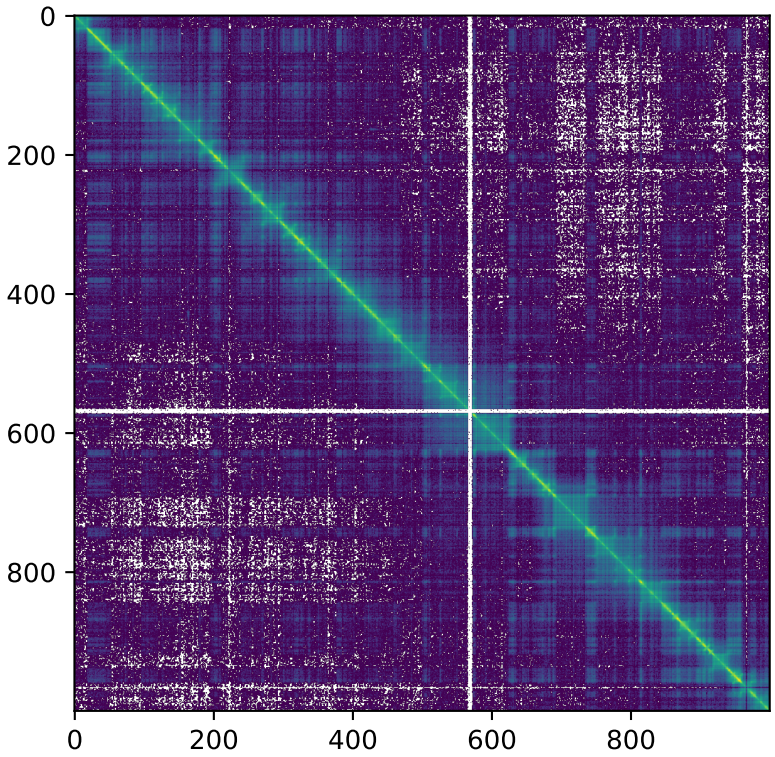
```
In [13]: sel = f.fetch("2L:10,000,000-20,000,000", join=True)

In [14]: sel.to_df()
Out[14]:
       chrom1    start1      end1 chrom2    start2      end2  count
0          2L  10000000  10010000     2L  10000000  10010000   6759
1          2L  10000000  10010000     2L  10010000  10020000   3241
2          2L  10000000  10010000     2L  10020000  10030000    760
3          2L  10000000  10010000     2L  10030000  10040000    454
4          2L  10000000  10010000     2L  10040000  10050000    289
...       ...       ...       ...    ...       ...       ...    ...
339036     2L  19970000  19980000     2L  19980000  19990000    407
339037     2L  19970000  19980000     2L  19990000  20000000    221
339038     2L  19980000  19990000     2L  19980000  19990000    391
339039     2L  19980000  19990000     2L  19990000  20000000    252
339040     2L  19990000  20000000     2L  19990000  20000000    266

[339041 rows x 7 columns]
```

3.3.2 Fetching interactions as scipy.sparse.csr_matrix

```
In [15]: sel = f.fetch("2L:10,000,000-20,000,000")

In [16]: sel.to_csr()
Out[16]:
<Compressed Sparse Row sparse matrix of dtype 'int32'
        with 339041 stored elements and shape (1000, 1000)>
```

3.3.3 Fetching interactions as numpy NDArray

```
In [17]: sel = f.fetch("2L:10,000,000-20,000,000")

In [18]: m = sel.to_numpy()

In [19]: import matplotlib.pyplot as plt

In [20]: from matplotlib.colors import LogNorm

In [21]: plt.imshow(m, norm=LogNorm())

In [22]: plt.show()
```



3.4 Fetching other types of data

Fetching the table of bins as pandas.DataFrame:

```
In [23]: f.bins()
Out[23]:
      chrom    start      end
0        2L        0    10000
1        2L    10000    20000
2        2L    20000    30000
3        2L    30000    40000
4        2L    40000    50000
...     ...      ...      ...
13753     Y  3620000  3630000
13754     Y  3630000  3640000
13755     Y  3640000  3650000
13756     Y  3650000  3660000
13757     Y  3660000  3667352

[13758 rows x 3 columns]
```

Fetching balancing weights:

```
In [24]: import pandas as pd

In [25]: weights = {}
    ...: for norm in f.avail_normalizations():
    ...:     weights[norm] = f.weights(norm)
    ...: weights = pd.DataFrame(weights)
    ...: weights
Out[25]:
             KR        VC   VC_SQRT
0      0.582102  0.666016  0.759389
1      1.300415  1.496604  1.138349
2      1.180977  1.470464  1.128364
3      1.007625  1.266340  1.047122
4      1.175642  1.492664  1.136850
...         ...       ...       ...
13753       NaN  0.000000  0.000000
13754       NaN  0.000000  0.000000
13755       NaN  0.000000  0.000000
13756  1.155544  2.234906  0.631055
13757       NaN  0.069841  0.111556

[13758 rows x 3 columns]
```

3.5 Efficiently compute descriptive statistics

hictkpy supports computing common descriptive statistics without reading interactions into memory (and without traversing the data more than once).

Compute all supported statistics at once:

```
In [26]: f.fetch().describe()
Out[26]:
{'nnz': 18122793,
 'sum': 114355295,
 'min': 1,
 'max': 53908,
 'mean': 6.310025998751958,
 'variance': 9918.666837525623,
```

(continues on next page)

(continued from previous page)

```
 'skewness': 83.28386530442891,
 'kurtosis': 20043.612488253475}
```

For more details, please refer to the **Statistics** section of the API docs for the *hictkpy.PixelSelector* class.

CHAPTER

FOUR

CREATING .COOL AND .HIC FILES

hictkpy supports creating .cool and .hic files from pre-binned interactions in COO or BedGraph2 format.

The examples in this section use file 4DNFIOTPSS3L.hic, which can be downloaded from the 4D Nucleome Data Portal here.

4.1 Preparation

The first step involves converting interactions from `4DNFIOTPSS3L.hic` to bedGraph2 format. This can be achieved using `hictk dump` (or alternatively with *`hictkpy.File.fetch()`*).

```
user@dev:/tmp$ hictk dump --join 4DNFIOTPSS3L.hic --resolution 50000 > pixels.bg2

user@dev:/tmp$ head pixels.bg2

  2L  0       50000   2L      0       50000   30211
  2L  0       50000   2L      50000   100000  13454
  2L  0       50000   2L      100000  150000  2560
  2L  0       50000   2L      150000  200000  911
  2L  0       50000   2L      200000  250000  753
  2L  0       50000   2L      250000  300000  846
  2L  0       50000   2L      300000  350000  530
  2L  0       50000   2L      350000  400000  378
  2L  0       50000   2L      400000  450000  630
  2L  0       50000   2L      450000  500000  756
```

Next, we also generate the list of chromosomes to use as reference.

```
user@dev:/tmp$ hictk dump -t chroms 4DNFIOTPSS3L.hic > chrom.sizes

user@dev:/tmp$ head chrom.sizes

  2L  23513712
  2R  25286936
  3L  28110227
  3R  32079331
  4   1348131
  X   23542271
  Y   3667352
```

4.2 Ingesting interactions in a .cool file

```
In [1]: import hictkpy as htk
```

(continues on next page)

(continued from previous page)

```
In [2]: import pandas as pd

# Create a dictionary mapping chromosome names to chromosome sizes
In [3]: chroms = pd.read_table("chrom.sizes", names=["name", "length"])
   ...:            .set_index("name")["length"]
   ...:            .to_dict()

In [4]: chroms
Out[4]:
{'2L': 23513712,
 '2R': 25286936,
 '3L': 28110227,
 '3R': 32079331,
 '4': 1348131,
 'X': 23542271,
 'Y': 3667352}

# Define the name of the columns for later use
In [5]: cols = ["chrom1", "start1", "end1",
   ...:         "chrom2", "start2", "end2",
   ...:         "count"]

# Initialize an empty .cool file
In [6]: with htk.cooler.FileWriter("out.cool", chroms, resolution=50_000) as writer:
   ...:     # Lazily load pixels in chunks to reduce memory usage
   ...:     pixels = pd.read_table("pixels.bg2", names=cols, chunksize=1_000_000)
   ...:     # Add chunks of pixels one by one
   ...:     for i, df in enumerate(pixels):
   ...:         print(f"adding chunk #{i}...")
   ...:         writer.add_pixels(df)
   ...:
adding chunk #0...
adding chunk #1...
adding chunk #2...
adding chunk #3...

# Check that the resulting file has some interactions
In [7]: htk.File("out.cool").attributes()["nnz"]
Out[7]: 3118456
```

4.3 Ingesting interactions in a .hic file

Follow the same steps as above for `.cool` files, but replace `htk.cooler.FileWriter` with `htk.hic.FileWriter`.

4.4 Tips and tricks

When loading interactions into a .cool or .hic file, interactions are initially stored in a temporary file. For a large number of interactions, this temporary file can become quite large. In such cases, it may be appropriate to pass a custom temporary folder where these files will be created:

```
In [1]: f = htk.cooler.FileWriter("out.cool", chroms, resolution=50_000, tmpdir="/var/
↪tmp/hictk")
```

When ingesting interactions in a .hic file, performance can be improved by using multiple threads:

```
In [1]: f = htk.hic.FileWriter("out.hic", chroms, resolution=50_000, n_threads=8)
```

When memory allows, it is possible to bypass temporary file creation by specifying a very large chunk size and ingesting all interactions at once. This can significantly speed up file creation:

```
# Initialize an empty .cool file

In [1]: cols = ["chrom1", "start1", "end1",
   ...:         "chrom2", "start2", "end2",
   ...:         "count"]

In [2]: df = pd.read_table("pixels.bg2", names=cols)

In [3]: with htk.cooler.FileWriter("out.cool", chroms, resolution=50_000, chunk_
↪size=len(df) + 1) as writer:
   ...:     writer.add_pixels(df)
   ...:
```

In case it is not possible to install a compatible version of pandas or pyarrow, the `FileWriter` classes support ingesting interactions from dictionaries of iterables (e.g., a dictionary mapping keys `bin1_id`, `bin2_id`, and `count` to iterables yielding numbers of the appropriate type).

For more details, refer to the documentation for *hictkpy.cooler.FileWriter.add_pixels_from_dict()* and *hictkpy.hic.FileWriter.add_pixels_from_dict()*.

CHAPTER

FIVE

MULTITHREADING AND MULTIPROCESSING

This section outlines how to best take advantage of multithreading and multiprocessing when using hictkpy.

5.1 TLDR

- Never share file handles, pixel selectors, or iterators between Python threads or processes.
- Using multithreading when fetching interactions from Cooler files rarely yields performance improvements. This is due to limitations of the HDF5 library itself.
- When using multiprocessing on Linux, avoid using fork as the method to start new processes. When using forkserver, avoid pre-loading the *hictkpy* library with `multiprocessing.set_forkserver_preload()`.
- Free-threaded Python is supported, in the sense that it is possible to build a wheel targeting free-threaded builds, but the resulting wheels still rely on the GIL.

5.2 Multithreading

Generally speaking, multithreaded code in Python cannot be used to improve performance for compute-bound operations. This is due to the Python GIL. However, the core of hictkpy is written in C++ and interacts with Python through the C API, which allows us to release the GIL on most long-running operations, such as fetching pixels from Cooler or .hic files. This allows applications to achieve concurrency through multithreading.

Furthermore, certain operations, such as creating .hic files, can natively take advantage of multicore CPUs by using multithreading. These threads are C++ threads and are completely independent from Python and the GIL.

It should be noted that within-process concurrency is limited when processing Cooler files, as HDF5, the C library used for low-level IO, makes heavy use of global state that is either not thread-safe or is protected by a global mutex.

5.3 Multiprocessing

In a multiprocessing environment, *hictkpy* behaves like any other Python library. The only area requiring special attention is logging. Logging is not supported when using fork as the method to start new processes on Linux, and will result in a `UserWarning` being raised.

```
/usr/lib64/python3.14/multiprocessing/popen_fork.py:70: UserWarning: hictkpy:␣
↪detected a call to fork():
hictkpy's logger does not support multiprocessing when using fork() as start method.
Please change process start method to spawn or forkserver.
For more details, refer to Python's documentation:
https://docs.python.org/3/library/multiprocessing.html#multiprocessing.set_start_
↪method
  self.pid = os.fork()
```

When using multiprocessing, processing Cooler files is not affected by the limitations described in the previous section.

5.4 Example (multithreading)

This example shows how to correctly use *hictkpy.File()* using concurrent.futures.ThreadPoolExecutor:

```
import logging
import os
import sys
import threading
import time
from concurrent.futures import ThreadPoolExecutor

import hictkpy
import pandas as pd


def fetch_chroms(path):
    """
    Get the list of chromosomes available in the given file
    """
    with hictkpy.MultiResFile(path) as f:
        return list(f.chromosomes().keys())


def fetch_pixels(path, resolution, query):
    """
    Fetch interactions for the given query and return them as a pandas.DataFrame
    """
    with hictkpy.File(path, resolution) as f:
        logging.info("[%s; TID=%d]: fetching...", query, threading.get_native_id())
        df = f.fetch(query, join=True).to_df()
        logging.info("[%s; TID=%d]: fetched %d interactions!", query, threading.get_
↪native_id(), len(df))

        return df


def fetch_cis_interactions(path, resolution, nthreads):
    """
    Fetch cis interactions from the given file and return them as a pandas.DataFrame
    """
    chroms = fetch_chroms(path)

    with ThreadPoolExecutor(nthreads) as tpool:
        tasks = []
        for chrom in chroms:
            tasks.append(tpool.submit(fetch_pixels, path, resolution, chrom))

        results = (task.result() for task in tasks)

        return pd.concat((df for df in results if len(df) != 0))


def setup_logger(level=logging.INFO):
```

(continues on next page)

(continued from previous page)

```
    fmt = "[%(asctime)s] %(levelname)s: %(message)s"
    logging.basicConfig(format=fmt)
    logging.getLogger().setLevel(level)

    # suppress log messages generated by hictkpy for level INFO or lower
    hictkpy.logging.setLevel(logging.WARN)


def main():
    setup_logger()

    path = "test/data/hic_test_file.hic"
    resolution = 100_000

    t0 = time.time()
    df = fetch_cis_interactions(path, resolution, nthreads=os.cpu_count())
    print(df, file=sys.stderr)

    logging.info("fetched %d interactions in %.2fs!", len(df), time.time() - t0)


if __name__ == "__main__":
    main()
```

5.5 Example (multiprocessing)

Using *hictkpy.File()* handles with using concurrent.futures.ProcessPoolExecutor is almost identical to the previous example.

- Explicitly set the process start method to something other than `fork` (only required on Linux).
- Initialize the logger in each child process by passing `initializer=setup_logger` to concurrent.futures.ProcessPoolExecutor.

```
import logging
import multiprocessing as mp
import os
import sys
import time
from concurrent.futures import ProcessPoolExecutor

import hictkpy
import pandas as pd


def fetch_chroms(path):
    """
    Get the list of chromosomes available in the given file
    """
    with hictkpy.MultiResFile(path) as f:
        return list(f.chromosomes().keys())


def fetch_pixels(path, resolution, query):
    """
    Fetch interactions for the given query and return them as a pandas.DataFrame
```

(continues on next page)

(continued from previous page)

```
    """
    with hictkpy.File(path, resolution) as f:
        logging.info("[%s; PID=%d]: fetching...", query, os.getpid())
        df = f.fetch(query, join=True).to_df()
        logging.info("[%s; PID=%d]: fetched %d interactions!", query, os.getpid(),␣
↪len(df))

        return df


def fetch_cis_interactions(path, resolution, nthreads):
    """
    Fetch cis interactions from the given file and return them as a pandas.DataFrame
    """
    chroms = fetch_chroms(path)

    with ProcessPoolExecutor(nthreads, initializer=setup_logger) as ppool:
        tasks = []
        for chrom in chroms:
            tasks.append(ppool.submit(fetch_pixels, path, resolution, chrom))

        results = (task.result() for task in tasks)

        return pd.concat((df for df in results if len(df) != 0))


def setup_logger(level=logging.INFO):
    fmt = "[%(asctime)s] %(levelname)s: %(message)s"
    logging.basicConfig(format=fmt)
    logging.getLogger().setLevel(level)

    # suppress log messages generated by hictkpy for level INFO or lower
    hictkpy.logging.setLevel(logging.WARN)


def main():
    mp.set_start_method("spawn")
    setup_logger()

    path = "test/data/hic_test_file.hic"
    resolution = 100_000

    t0 = time.time()
    df = fetch_cis_interactions(path, resolution, nthreads=os.cpu_count())
    print(df, file=sys.stderr)

    logging.info("fetched %d interactions in %.2fs!", len(df), time.time() - t0)


if __name__ == "__main__":
    main()
```

CHAPTER

SIX

PYTHON API REFERENCE

This section provides a detailed reference for the `hictkpy` Python API.

The API is organized in three categories:

- *Generic API* – Documents classes and functions such as *`hictkpy.File`* and *`hictkpy.PixelSelector`* that are used to open, inspect, and fetch interactions from `.[m]cool` and `.hic` files. The documentation also covers several general-purpose classes and data structures such as *`hictkpy.BinTable`*, *`hictkpy.Bin`*, and *`hictkpy.Pixel`* which are utilized across the API.
- *Cooler API* – Documents the *`hictkpy.cooler.SingleCellFile`* and *`hictkpy.cooler.FileWriter`* classes, which are used to access `.scool` files and create `.cool` files, respectively.
- *.hic API* – Documents the *`hictkpy.hic.FileWriter`* class, which is used to create `.hic` files.
- *Logging API* – Documents how to tweak hictkpy's `logging.Logger`.

6.1 Generic API

hictkpy.is_cooler(*path: str | PathLike*) → bool

Test whether path points to a cooler file.

hictkpy.is_mcool_file(*path: str | PathLike*) → bool

Test whether path points to a .mcool file.

hictkpy.is_scool_file(*path: str | PathLike*) → bool

Test whether path points to a .scool file.

hictkpy.is_hic(*path: str | PathLike*) → bool

Test whether path points to a .hic file.

class hictkpy.**MultiResFile**(**args*, ***kwargs*)

Class representing a file handle to a .hic or .mcool file

__init__(*self*, *path: str | PathLike*) → None

Open a multi-resolution Cooler file (.mcool) or .hic file.

__getitem__(*self*, *arg: int*, */ (Positional-only parameter separator (PEP 570))*) → *File*

Open the Cooler or .hic file corresponding to the resolution given as input.

__enter__(*self*) → *MultiResFile*

__exit__(
self,
exc_type: object | None = None,
exc_value: object | None = None,
traceback: object | None = None,
) → None

attributes(*self*) → dict

Get file attributes as a dictionary.

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get the chromosome sizes as a dictionary mapping names to sizes.

close(*self*) → None

Manually close the file handle.

is_hic(*self*) → bool

Test whether the file is in .hic format.

is_mcool(*self*) → bool

Test whether the file is in .mcool format.

path(*self*) → Path

Get the file path.

resolutions(
self,
) → numpy.ndarray[dtype=int64, shape=(*), order='C']

Get the list of available resolutions.

class hictkpy.**File**(**args*, ***kwargs*)

Class representing a file handle to a .cool or .hic file.

__init__(
self,
path: str | PathLike,
resolution: int | None = None,
matrix_type: str = 'observed',
matrix_unit: str = 'BP',
) → None

Construct a file object to a .hic, .cool or .mcool file given the file path and resolution. Resolution is ignored when opening single-resolution Cooler files.

__enter__(*self*) → *File*

__exit__(
self,
exc_type: object | None = None,
exc_value: object | None = None,
traceback: object | None = None,
) → None

attributes(*self*) → dict

Get file attributes as a dictionary.

avail_normalizations(*self*) → list[str]

Get the list of available normalizations.

bins(*self*) → *BinTable*

Get table of bins.

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get chromosome sizes as a dictionary mapping names to sizes.

close(*self*) → None

Manually close the file handle.

fetch(
self,
range1: str | None = None,
range2: str | None = None,
normalization: str | None = None,
count_type: type | str = 'int32',
join: bool = False,
query_type: str = 'UCSC',
diagonal_band_width: int | None = None,
) → *PixelSelector*

Fetch interactions overlapping a region of interest.

has_normalization(*self*, *normalization: str*) → bool

Check whether a given normalization is available.

is_cooler(*self*) → bool

Test whether file is in .cool format.

is_hic(*self*) → bool

Test whether file is in .hic format.

nbins(*self*) → int

Get the total number of bins.

nchroms(*self*, *include_ALL: bool = False*) → int

Get the total number of chromosomes.

path(*self*) → Path

Return the file path.

resolution(*self*) → int

Get the bin size in bp.

uri(*self*) → str

Return the file URI.

weights(
self,
name: str,
divisive: bool = True,
) → numpy.ndarray[dtype=float64, shape=(*), order='C'] | None

weights(
self,
names: Sequence[str],
divisive: bool = True,
) → DataFrame

Overloaded function.

1. `weights(self, name: str, divisive: bool = True) -> numpy.ndarray[dtype=float64, shape=(*), order='C'] | None`

Fetch the balancing weights for the given normalization method.

2. `weights(self, names: collections.abc.Sequence[str], divisive: bool = True) -> pandas.DataFrame`

Fetch the balancing weights for the given normalization methods.Weights are returned as a pandas.DataFrame.

class hictkpy.**PixelSelector**(**args*, ***kwargs*)

Class representing pixels overlapping with the given genomic intervals.

coord1(*self*) → tuple[str, int, int] | None

Get query coordinates for the first dimension. Returns None when query spans the entire genome.

coord2(*self*) → tuple[str, int, int] | None

Get query coordinates for the second dimension. Returns None when query spans the entire genome.

dtype(*self*) → type

Get the dtype for the pixel count.

to_arrow(
self,
query_span: str = 'upper_triangle',
) → Table

Retrieve interactions as a pyarrow.Table.

to_coo(
self,
query_span: str = 'upper_triangle',
low_memory: bool = False,
) → coo_matrix

Retrieve interactions as a SciPy COO matrix. When low_memory=True, the heuristic used to minimize the number of memory allocations is turned off, and a two-pass algorithm that allocates a matrix with the exact shape is used instead.

to_csr(
self,
query_span: str = 'upper_triangle',
low_memory: bool = False,
) → csr_matrix

Retrieve interactions as a SciPy CSR matrix. When low_memory=True, the heuristic used to minimize the number of memory allocations is turned off, and a two-pass algorithm that allocates a matrix with the exact shape is used instead.

to_df(
self,
query_span: str = 'upper_triangle',
) → DataFrame

Alias to to_pandas().

to_numpy(
self,
query_span: str = 'full',
) → numpy.ndarray[shape=(*, *), order='C']

Retrieve interactions as a numpy 2D matrix.

to_pandas(
self,
query_span: str = 'upper_triangle',
) → DataFrame

Retrieve interactions as a pandas DataFrame.

size(*self*, *upper_triangular: bool = True*) → int

Get the number of pixels overlapping with the given query.

Statistics

hictkpy.PixelSelector exposes several methods to compute or estimate several statistics efficiently.

The main features of these methods are:

- All statistics are computed by traversing the data only once and without caching interactions.
- All methods can be tweaked to include or exclude non-finite values.

- All functions implemented using short-circuiting to detect scenarios where the required statistics can be computed without traversing all pixels.

The following statistics are guaranteed to be exact:

- nnz
- sum
- min
- max
- mean

The rest of the supported statistics (currently variance, skewness, and kurtosis) are estimated and are thus not guaranteed to be exact. However, in practice, the estimation is usually very accurate (relative error < 1.0e-6).

You can instruct hictkpy to compute the exact statistics by passing `exact=True` to *hictkpy.PixelSelector.describe()* and related methods. It should be noted that for large queries this will result in slower computations and higher memory usage.

describe(
self,
metrics: Sequence[str] = ['nnz', 'sum', 'min', 'max', 'mean', 'variance', 'skewness', 'kurtosis'],
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
exact: bool = False,
) → dict

Compute one or more descriptive metrics in the most efficient way possible. Known metrics: nnz, sum, min, max, mean, variance, skewness, kurtosis. When a metric cannot be computed (e.g. because metrics=["variance"], but selector overlaps with a single pixel), the value for that metric is set to None. When keep_infs or keep_nans are set to True, and keep_zeros=True, nan and/or inf values are treated as zeros. By default, metrics are estimated by doing a single pass through the data. The estimates are stable and usually very accurate. However, if you require exact values, you can specify exact=True.

kurtosis(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
exact: bool = False,
) → float | None

Get the kurtosis of the number of interactions for the current pixel selection. See documentation for describe() for more details.

max(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
) → int | float | None

Get the maximum number of interactions for the current pixel selection. See documentation for describe() for more details.

mean(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
) → float | None

Get the average number of interactions for the current pixel selection. See documentation for describe() for more details.

min(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
) → int | float | None

Get the minimum number of interactions for the current pixel selection. See documentation for describe() for more details.

nnz(*self*, *keep_nans: bool = False*, *keep_infs: bool = False*) → int

Get the number of non-zero entries for the current pixel selection. See documentation for describe() for more details.

skewness(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
exact: bool = False,
) → float | None

Get the skewness of the number of interactions for the current pixel selection. See documentation for describe() for more details.

sum(
self,
keep_nans: bool = False,
keep_infs: bool = False,
) → int | float

Get the total number of interactions for the current pixel selection. See documentation for describe() for more details.

variance(
self,
keep_nans: bool = False,
keep_infs: bool = False,
keep_zeros: bool = False,
exact: bool = False,
) → float | None

Get the variance of the number of interactions for the current pixel selection. See documentation for describe() for more details.

Iteration

__iter__(*self*) → hictkpy.PixelIterator

Implement iter(self). The resulting iterator yields objects of type hictkpy.Pixel.

```
In [1]: import hictkpy as htk

In [2]: f = htk.File("file.cool")

In [3]: sel = f.fetch("chr2L:10,000,000-20,000,000")

In [4]: for i, pixel in enumerate(sel):
   ...:     print(pixel.bin1_id, pixel.bin2_id, pixel.count)
   ...:     if i > 10:
   ...:         break
```

(continues on next page)

(continued from previous page)

```
   ...:
1000 1000 6759
1000 1001 3241
1000 1002 760
1000 1003 454
1000 1004 289
1000 1005 674
1000 1006 354
1000 1007 124
1000 1008 130
1000 1009 105
1000 1010 99
1000 1011 120
```

It is also possible to iterate over pixels together with their genomic coordinates by specifying `join=True` when calling *hictkpy.File.fetch()*:

```
In [5]: sel = f.fetch("chr2L:10,000,000-20,000,000", join=True)

In [6]: for i, pixel in enumerate(sel):
   ...:     print(
   ...:         pixel.chrom1, pixel.start1, pixel.end1,
   ...:         pixel.chrom2, pixel.start2, pixel.end2,
   ...:         pixel.count
   ...:     )
   ...:     if i > 10:
   ...:         break
   ...:
chr2L 10000000 10010000 chr2L 10000000 10010000 6759
chr2L 10000000 10010000 chr2L 10010000 10020000 3241
chr2L 10000000 10010000 chr2L 10020000 10030000 760
chr2L 10000000 10010000 chr2L 10030000 10040000 454
chr2L 10000000 10010000 chr2L 10040000 10050000 289
chr2L 10000000 10010000 chr2L 10050000 10060000 674
chr2L 10000000 10010000 chr2L 10060000 10070000 354
chr2L 10000000 10010000 chr2L 10070000 10080000 124
chr2L 10000000 10010000 chr2L 10080000 10090000 130
chr2L 10000000 10010000 chr2L 10090000 10100000 105
chr2L 10000000 10010000 chr2L 10100000 10110000 99
chr2L 10000000 10010000 chr2L 10110000 10120000 120
```

class hictkpy.**Bin**

Class representing a genomic Bin (i.e., a BED interval).

property id

Get the bin ID.

property rel_id

Get the relative bin ID (i.e., the ID that uniquely identifies a bin within a chromosome).

property chrom

Get the name of the chromosome to which the Bin refers to.

property start

Get the Bin start position.

property end

Get the Bin end position.

class hictkpy.**BinTable**(**args*, ***kwargs*)

Class representing a table of genomic bins.

__init__(*self*, *chroms: dict[str, int]*, *resolution: int*) → None

__init__(*self*, *bins: DataFrame*) → None

Overloaded function.

1. `__init__(self, chroms: dict[str, int], resolution: int) -> None`

Construct a table of bins given a dictionary mapping chromosomes to their sizes and a resolution.

2. `__init__(self, bins: pandas.DataFrame) -> None`

Construct a table of bins from a pandas.DataFrame with columns ["chrom", "start", "end"].

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get the chromosome sizes as a dictionary mapping names to sizes.

get(*self*, *bin_id: int*) → *Bin*

get(*self*, *bin_ids: Sequence[int]*) → DataFrame

get(*self*, *chrom: str*, *pos: int*) → *Bin*

get(
self,
chroms: Sequence[str],
pos: Sequence[int],
) → DataFrame

Overloaded function.

1. `get(self, bin_id: int) -> hictkpy.Bin`

Get the genomic coordinate given a bin ID.

2. `get(self, bin_ids: collections.abc.Sequence[int]) -> pandas.DataFrame`

Get the genomic coordinates given a sequence of bin IDs. Genomic coordinates are returned as a pandas.DataFrame with columns ["chrom", "start", "end"].

3. `get(self, chrom: str, pos: int) -> hictkpy.Bin`

Get the bin overlapping the given genomic coordinate.

4. `get(self, chroms: collections.abc.Sequence[str], pos: collections.abc.Sequence[int]) -> pandas.DataFrame`

Get the bins overlapping the given genomic coordinates. Bins are returned as a pandas.DataFrame with columns ["chrom", "start", "end"].

get_id(*self*, *chrom: str*, *pos: int*) → int

Get the ID of the bin overlapping the given genomic coordinate.

get_ids(
self,
chroms: Sequence[str],
pos: Sequence[int],
) → numpy.ndarray[dtype=int64, shape=(*)]

Get the IDs of the bins overlapping the given genomic coordinates.

merge(*self*, *df: DataFrame*) → DataFrame

Merge genomic coordinates corresponding to the given bin identifiers. Bin identifiers should be provided as a pandas.DataFrame with columns "bin1_id" and "bin2_id". Genomic coordinates are returned as a pandas.DataFrame containing the same data as the DataFrame given as input, plus columns ["chrom1", "start1", "end1", "chrom2", "start2", "end2"].

resolution(*self*) → int

Get the bin size for the bin table. Return 0 in case the bin table has a variable bin size.

to_arrow(
self,
range: str | None = None,
query_type: str = 'UCSC',
) → Table

Return the bins in the BinTable as a pyarrow.Table. The optional "range" parameter can be used to only fetch a subset of the bins in the BinTable.

to_df(
self,
range: str | None = None,
query_type: str = 'UCSC',
) → DataFrame

Alias to to_pandas().

to_pandas(
self,
range: str | None = None,
query_type: str = 'UCSC',
) → DataFrame

Return the bins in the BinTable as a pandas.DataFrame. The optional "range" parameter can be used to only fetch a subset of the bins in the BinTable.

type(*self*) → str

Get the type of table underlying the BinTable object (i.e. fixed or variable).

__iter__(*self*) → hictkpy.BinTableIterator

Implement iter(self). The resulting iterator yields objects of type hictkpy.Bin.

class hictkpy.**Pixel**(**args*, ***kwargs*)

Class modeling a Pixel in COO or BG2 format.

property bin1_id

Get the ID of bin1.

property bin2_id

Get the ID of bin2.

property count

Get the number of interactions.

The following properties are only available when pixels are in BG2 format.

property bin1

Get bin1.

property bin2

Get bin2.

property chrom1

Get the chromosome associated with bin1.

property start1

Get the start position associated with bin1.

property end1

Get the end position associated with bin1.

property chrom2

Get the chromosome associated with bin2.

property start2

Get the start position associated with bin2.

property end2

Get the end position associated with bin2.

6.2 Logging API

hictkpy.logging.**setLevel**(*level: int | str*) → None

Test the log level for hictkpy's logger. Accepts the predefined levels defined by the logging module.

Using this method instead should be preferred to tweak the log level with `logging.getLogger("hictkpy").setLevel("INFO")`, as *`hictkpy.logging.setLevel()`* sets the log level also on the underlying the C++ logger, thus avoiding needlessly producing log messages on the C++ side to then discard them once they reach the Python logger.

6.3 Cooler API

class hictkpy.cooler.**SingleCellFile**(**args*, ***kwargs*)

Class representing a file handle to a .scool file.

__init__(*self*, *path: str | PathLike*) → None

Open a single-cell Cooler file (.scool).

__getitem__(*self*, *cell_id: str*) → *File*

Open the Cooler file corresponding to the cell ID given as input.

__enter__(*self*) → *SingleCellFile*

__exit__(
self,
exc_type: object | None = None,
exc_value: object | None = None,
traceback: object | None = None,
) → None

attributes(*self*) → dict

Get file attributes as a dictionary.

bins(*self*) → *BinTable*

Get table of bins.

cells(*self*) → list[str]

Get the list of available cells.

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get the chromosome sizes as a dictionary mapping names to sizes.

close(*self*) → None

Manually close the file handle.

path(*self*) → Path

Get the file path.

resolution(*self*) → int

Get the bin size in bp.

class hictkpy.cooler.**FileWriter**(**args*, ***kwargs*)

Class representing a file handle to create .cool files.

__init__(
self,
path: str | PathLike,
chromosomes: dict[str, int],
resolution: int,
assembly: str = 'unknown',
tmpdir: str | PathLike = PosixPath('/tmp'),
compression_lvl: int = 6,
) → None

__init__(
self,
path: str | PathLike,
bins: BinTable,
assembly: str = 'unknown',
tmpdir: str | PathLike = PosixPath('/tmp'),
compression_lvl: int = 6,
) → None

Overloaded function.

1. `__init__(self, path: str | os.PathLike, chromosomes: dict[str, int], resolution: int, assembly: str = 'unknown', tmpdir: str | os.PathLike = PosixPath('/tmp'), compression_lvl: int = 6) -> None`

   Open a .cool file for writing given a list of chromosomes with their sizes and a resolution.

2. `__init__(self, path: str | os.PathLike, bins: hictkpy._hictkpy.BinTable, assembly: str = 'unknown', tmpdir: str | os.PathLike = PosixPath('/tmp'), compression_lvl: int = 6) -> None`

   Open a .cool file for writing given a table of bins.

__enter__(*self*) → *FileWriter*

__exit__(
self,
exc_type: object | None = None,
exc_value: object | None = None,
traceback: object | None = None,
) → None

add_pixels(
self,
pixels: DataFrame | Table,
sorted: bool = False,
validate: bool = True,
) → None

Add pixels from a pandas.DataFrame or pyarrow.Table containing pixels in COO or BG2 format (i.e. either with columns=[bin1_id, bin2_id, count] or with columns=[chrom1, start1, end1, chrom2, start2, end2, count]). When sorted is True, pixels are assumed to be sorted by their genomic coordinates in ascending order. When validate is True, hictkpy will perform some basic sanity checks on the given pixels before adding them to the Cooler file.

add_pixels_from_dict(
self,
columns: Dict[str, Iterable[str | int | float]],
sorted: bool = False,
validate: bool = True,
) → None

Add pixels from a dictionary containing containing columns corresponding to pixels in COO or BG2 format (i.e. either with keys=[bin1_id, bin2_id, count] or with keys=[chrom1, start1, end1, chrom2, start2, end2, count]). When sorted is True, pixels are assumed to be sorted by their genomic coordinates in ascending order. When validate is True, hictkpy will perform some basic sanity checks on the given pixels before adding them to the Cooler file.

bins(*self*) → *BinTable*

Get table of bins.

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get the chromosome sizes as a dictionary mapping names to sizes.

finalize(
self,
log_lvl: str | None = None,
chunk_size: int = 500000,
update_frequency: int = 10000000,
) → *File*

Write interactions to file.

path(*self*) → Path

Get the file path.

resolution(*self*) → int

Get the resolution in bp.

6.4 .hic API

class hictkpy.hic.**FileWriter**(**args*, ***kwargs*)

Class representing a file handle to create .hic files.

__init__(
self,
path: str | PathLike,
chromosomes: dict[str, int],
resolution: int,
assembly: str = 'unknown',
n_threads: int = 1,
chunk_size: int = 10000000,
tmpdir: str | PathLike = PosixPath('/tmp'),
compression_lvl: int = 10,
skip_all_vs_all_matrix: bool = False,
) → None

__init__(
self,
path: str | PathLike,
chromosomes: dict[str, int],
resolutions: Sequence[int],
assembly: str = 'unknown',
n_threads: int = 1,
chunk_size: int = 10000000,
tmpdir: str | PathLike = PosixPath('/tmp'),
compression_lvl: int = 10,
skip_all_vs_all_matrix: bool = False,
) → None

__init__(
self,

path: str | PathLike,
bins: BinTable,
assembly: str = 'unknown',
n_threads: int = 1,
chunk_size: int = 10000000,
tmpdir: str | PathLike = PosixPath('/tmp'),
compression_lvl: int = 10,
skip_all_vs_all_matrix: bool = False,
) → None

Overloaded function.

1. `__init__(self, path: str | os.PathLike, chromosomes: dict[str, int], resolution: int, assembly: str = 'unknown', n_threads: int = 1, chunk_size: int = 10000000, tmpdir: str | os.PathLike = PosixPath('/tmp'), compression_lvl: int = 10, skip_all_vs_all_matrix: bool = False) -> None`

Open a .hic file for writing given a list of chromosomes with their sizes and one resolution.

2. `__init__(self, path: str | os.PathLike, chromosomes: dict[str, int], resolutions: collections.abc.Sequence[int], assembly: str = 'unknown', n_threads: int = 1, chunk_size: int = 10000000, tmpdir: str | os.PathLike = PosixPath('/tmp'), compression_lvl: int = 10, skip_all_vs_all_matrix: bool = False) -> None`

Open a .hic file for writing given a list of chromosomes with their sizes and one or more resolutions.

3. `__init__(self, path: str | os.PathLike, bins: hictkpy._hictkpy.BinTable, assembly: str = 'unknown', n_threads: int = 1, chunk_size: int = 10000000, tmpdir: str | os.PathLike = PosixPath('/tmp'), compression_lvl: int = 10, skip_all_vs_all_matrix: bool = False) -> None`

Open a .hic file for writing given a BinTable. Only BinTable with a fixed bin size are supported.

__enter__(*self*) → *FileWriter*

__exit__(
self,
exc_type: object | None = None,
exc_value: object | None = None,
traceback: object | None = None,
) → None

add_pixels(
self,
pixels: DataFrame | Table,
validate: bool = True,
) → None

Add pixels from a pandas.DataFrame or pyarrow.Table containing pixels in COO or BG2 format (i.e. either with columns=[bin1_id, bin2_id, count] or with columns=[chrom1, start1, end1, chrom2, start2, end2, count]). When sorted is True, pixels are assumed to be sorted by their genomic coordinates in ascending order. When validate is True, hictkpy will perform some basic sanity checks on the given pixels before adding them to the .hic file.

add_pixels_from_dict(
self,
columns: Dict[str, Iterable[str | int | float]],
validate: bool = True,
) → None

Add pixels from a dictionary containing containing columns corresponding to pixels in COO or BG2 format (i.e. either with keys=[bin1_id, bin2_id, count] or with keys=[chrom1, start1, end1, chrom2,

start2, end2, count]). When sorted is True, pixels are assumed to be sorted by their genomic coordinates in ascending order. When validate is True, hictkpy will perform some basic sanity checks on the given pixels before adding them to the Cooler file.

bins(*self*, *resolution: int*) → *BinTable*

Get table of bins for the given resolution.

chromosomes(*self*, *include_ALL: bool = False*) → dict[str, int]

Get the chromosome sizes as a dictionary mapping names to sizes.

finalize(*self*, *log_lvl: str | None = None*) → *File*

Write interactions to file.

path(*self*) → Path

Get the file path.

resolutions(*self*) → numpy.ndarray[dtype=int64, shape=(*), order='C']

Get the list of resolutions in bp.

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

P

R

S

T

U

V

W