ngs-tools

Release 1.8.4

Kyung Hoi (Joseph) Min

Aug 16, 2023

CONTENTS:

CHAPTER

ONE

NGS_TOOLS

1.1 Subpackages

1.1.1 ngs_tools.binary

Submodules

ngs_tools.binary.Argument

Module Contents

Classes

Argument	Abstract class representing a single argument.
PositionalArgument	Abstract class representing a single argument.
ConstantArgument	Abstract class representing a single argument.
NamedArgument	Abstract class representing a single argument.

exception ngs_tools.binary.Argument.**ArgumentError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.binary.Argument.**Argument**(*name: str*, *pre_validator:* ngs_tools.binary.ArgumentValidator.ArgumentValidator = *NoValidator()*, *post_validator:* ngs_tools.binary.ArgumentValidator.ArgumentValidator = *NoValidator()*, *required: bool = True*)

Bases: abc.ABC

Abstract class representing a single argument.

property name: str

property required: bool

pre_execute(*arg: str*)

post_execute(*arg: str*)

render(*arg: str*) → List[str]

class ngs_tools.binary.Argument.**PositionalArgument**(*name: str*, *pre_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *post_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *required: bool = True*)

Bases: *Argument*

Abstract class representing a single argument.

class ngs_tools.binary.Argument.**ConstantArgument**(*name: str*, *pre_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *post_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *required: bool = True*)

Bases: *Argument*

Abstract class representing a single argument.

render(*arg: Any*) → List[str]

class ngs_tools.binary.Argument.**NamedArgument**(*name: str*, *pre_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *post_validator: ngs_tools.binary.ArgumentValidator.ArgumentValidator = NoValidator()*, *required: bool = True*)

Bases: *Argument*

Abstract class representing a single argument.

render(*arg: str*) → List[str]

ngs_tools.binary.ArgumentValidator

Module Contents

Classes

ArgumentValidator	Abstract argument validator.
AndValidator	Abstract argument validator.
OrValidator	Abstract argument validator.
NotValidator	Abstract argument validator.
NoValidator	Abstract argument validator.
IntegerValidator	Abstract argument validator.
FloatValidator	Abstract argument validator.
PositiveValidator	Abstract argument validator.
FileValidator	Abstract argument validator.
DirValidator	Abstract argument validator.
ConstantValidator	Abstract argument validator.

Attributes

IsInteger
IsFloat
IsPositive
IsPositiveInteger
IsDir
IsFile

class ngs_tools.binary.ArgumentValidator.**ArgumentValidator**

Bases: abc.ABC

Abstract argument validator.

abstract __call__(*arg*) → bool

__and__(*other:* ArgumentValidator) → *ArgumentValidator*

__or__(*other:* ArgumentValidator) → *ArgumentValidator*

__invert__() → *ArgumentValidator*

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**AndValidator**(*left:* ArgumentValidator, *right:* ArgumentValidator)

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**OrValidator**(*left:* ArgumentValidator, *right:* ArgumentValidator)

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**NotValidator**(*validator:* ArgumentValidator)

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**NoValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**IntegerValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**FloatValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**PositiveValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**FileValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**DirValidator**

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

class ngs_tools.binary.ArgumentValidator.**ConstantValidator**(*constant: str*)

Bases: *ArgumentValidator*

Abstract argument validator.

__call__(*arg*) → bool

__str__() → str

Return str(self).

ngs_tools.binary.ArgumentValidator.**IsInteger**

ngs_tools.binary.ArgumentValidator.**IsFloat**

ngs_tools.binary.ArgumentValidator.**IsPositive**

ngs_tools.binary.ArgumentValidator.**IsPositiveInteger**

ngs_tools.binary.ArgumentValidator.**IsDir**

ngs_tools.binary.ArgumentValidator.**IsFile**

ngs_tools.binary.Binary

Module Contents

Classes

BinaryResult	Typed version of namedtuple.
Binary	Wrapper around a binary file that provides a object-oriented interface to
BinaryExecutor	Class that handles execution of binaries with specified arguments.

exception ngs_tools.binary.Binary.**BinaryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.binary.Binary.**BinaryResult**

Bases: NamedTuple

Typed version of namedtuple.

Usage in Python versions >= 3.6:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has extra __annotations__ and _field_types attributes, giving an ordered dict mapping field names to types. __annotations__ should be preferred, while _field_types is kept to maintain pre PEP 526 compatibility. (The field names are in the _fields attribute, which is part of the namedtuple API.) Alternative equivalent keyword syntax is also accepted:

```
Employee = NamedTuple('Employee', name=str, id=int)
```

In Python versions <= 3.5 use:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

command: `List[str]`

process: `subprocess.Popen`

stdout: `Optional[str]`

stderr: `Optional[str]`

class ngs_tools.binary.Binary.**Binary**(*path: str*)

Wrapper around a binary file that provides a object-oriented interface to `run_executable()`.

property path: `str`

__str__() → str

Return str(self).

pre_execute(*command: str*, **args*, ***kwargs*)

post_execute(*command: str*, *result:* BinaryResult, **args*, ***kwargs*)

__call__(*command: List[str]*, **args*, ***kwargs*) → *BinaryResult*

Wrapper around the `run_executable()` function. The first element of the *command* argument to the function will always be the path to the current binary.

class ngs_tools.binary.Binary.**BinaryExecutor**(*binary:* Binary, **args*)

Class that handles execution of binaries with specified arguments.

pre_execute(*command: str*)

post_execute(*command: str*, *result:* BinaryResult)

__call__(*values: Dict[str, Optional[str]]*, **args*, ***kwargs*)

Run the binary with the specified values for each of the arguments.

1.1.2 ngs_tools.chemistry

Submodules

ngs_tools.chemistry.Chemistry

Module Contents

Classes

SubSequenceDefinition	Definition of a subsequence. This class is used to parse a subsequence out from
SubSequenceParser	Class that uses a collection of *SubSequenceDefinition* instances to parse
Chemistry	Base class to represent any kind of chemistry.
SequencingStrand	Generic enumeration.
SequencingChemistry	Base class to represent a sequencing chemistry.

Attributes

WHITELISTS_DIR

ngs_tools.chemistry.Chemistry.**WHITELISTS_DIR**

exception ngs_tools.chemistry.Chemistry.**SubSequenceDefinitionError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.Chemistry.**SubSequenceDefinition**(*index: int*, *start: Optional[int] = None*, *length: Optional[int] = None*)

Definition of a subsequence. This class is used to parse a subsequence out from a list of sequences.

TODO: anchoring

_index

Sequence index to use (from a list of sequences); for internal use only. Use *index* instead.

_start

Starting position of the subsequence; for internal use only. Use *start* instead.

_length

Length of the subsequence; for internal use only. Use `length` instead.

property index: int

Sequence index

property start: Optional[int]

Substring starting position

property end: Optional[int]

Substring end position. None if *start* or *length* is None.

property length: Optional[int]

Substring length. None if not provided on initialization.

is_overlapping(*other:* SubSequenceDefinition) → bool

Whether this subsequence overlaps with another subsequence.

Parameters

other – The other *SubSequenceDefinition* instance to compare to

Returns
True if they are overlapping. False otherwise.

parse(*s: List[str]*) → str

Parse the given list of strings according to the arguments used to initialize this instance. If *start* and *length* was not provided, then this is simply the entire string at index *index*. Otherwise, the substring from position *start* of length *length* is extracted from the string at index *index*.

Parameters
s – List of strings to parse

Returns
The parsed string

__eq__(*other:* SubSequenceDefinition)

Return self==value.

__repr__()

Return repr(self).

__str__()

Return str(self).

exception ngs_tools.chemistry.Chemistry.**SubSequenceParserError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.Chemistry.**SubSequenceParser**(**definitions:* SubSequenceDefinition)

Class that uses a collection of *SubSequenceDefinition* instances to parse an entire subsequence from a list of strings.

_definitions

List of *SubSequenceDefinition* instances; for internal use only.

property definitions

is_overlapping(*other:* SubSequenceParser) → bool

Whether this parser overlaps with another parser. Checks all pairwise combinations and returns True if any two *SubSequenceDefinition* instances overlap.

Parameters
other – The other *SubSequenceParser* instance to compare to

Returns
True if they are overlapping. False otherwise.

parse(*sequences: List[str]*, *concatenate: bool = False*) → Union[str, Tuple[str]]

Iteratively constructs a full subsequence by applying each *SubSequenceDefinition* in *_definitions* on the list of provided sequences. If `concatenate=False`, then this function returns a tuple of length equal to the number of definitions. Each element of the tuple is a string that was parsed by each definition. Otherwise, all the parsed strings are concatenated into a single string.

Parameters

- **sequences** – List of sequences to parse
- **concatenate** – Whether or not to concatenate the parsed strings. Defaults to False.

Returns
Concatenated parsed sequence (if `concatenate=True`). Otherwise, a tuple of parsed strings.

parse_reads(*reads: List[*ngs_tools.fastq.Read.Read*]*, *concatenate: bool = False*) → Tuple[Union[str, Tuple[str]], Union[str, Tuple[str]]]

Behaves identically to *parse()*, but instead on a list of *ngs_tools.fastq.Read* instances. *parse()* is called on the read sequences and qualities separately.

Parameters

- **reads** – List of reads to parse
- **concatenate** – Whether or not to concatenate the parsed strings. Defaults to False.

Returns

Parsed sequence from read sequences Parsed sequence from quality sequences

__eq__(*other:* SubSequenceParser)

Check whether this parser equals another. The order of definitions must also be equal.

__iter__()

__len__()

__getitem__(*i*)

__repr__()

Return repr(self).

__str__()

Return str(self).

exception ngs_tools.chemistry.Chemistry.**ChemistryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.Chemistry.**Chemistry**(*name: str*, *description: str*, *files: Optional[Dict[str, str]] = None*)

Bases: abc.ABC

Base class to represent any kind of chemistry.

_name

Chemistry name; for internal use only. Use *name* instead.

_description

Chemistry description; for internal use only. Use *description* instead.

_files

Dictionary containing files related to this chemistry. For internal use only.

property name: str

Chemistry name

property description: str

Chemistry description

has_file(*name: str*) → bool

Whether *_files* contains a file with the specified name

get_file(*name: str*) → bool

Get a file path by its name

class ngs_tools.chemistry.Chemistry.**SequencingStrand**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

UNSTRANDED = 0

FORWARD = 1

REVERSE

class ngs_tools.chemistry.Chemistry.**SequencingChemistry**(*n: int*, *strand:* SequencingStrand, *parsers: Dict[str,* SubSequenceParser*]*, **args*, ***kwargs*)

Bases: *Chemistry*

Base class to represent a sequencing chemistry.

property n: int

Number of sequences to parse at once

property parsers: Dict[str, *SubSequenceParser*]

Retrieve a copy of the `_parsers` dictionary.

property strand: *SequencingStrand*

Retrieve the strandedness of the chemistry.

property lengths: Tuple[int, Ellipsis]

The expected length for each sequence, based on *parsers*. *None* indicates any length is expected.

property has_barcode: bool

abstract property barcode_parser: *SubSequenceParser*

property has_umi: bool

abstract property umi_parser: *SubSequenceParser*

property has_whitelist: bool

abstract property whitelist_path

get_parser(*name: str*) → *SubSequenceParser*

Get a *SubSequenceParser* by its name

has_parser(*name: str*) → bool

Whether `_parsers` contains a parser with the specified name

reorder(*reordering: List[int]*) → *Chemistry*

Reorder the file indices according to the `reordering` list. This list reorders the file at each index to the value at that index.

Parameters

reordering – List containing how to reorder file indices, where the file at index `i` of this index will now be at index `reordering[i]`.

Returns

A new *Chemistry* instance (or the subclass)

parse(*sequences: List[str]*, *concatenate: bool = False*) → Dict[str, Union[str, Tuple[str]]]

Parse a list of strings using the parsers in `_parsers` and return a dictionary with keys corresponding to those in `_parsers`.

Parameters

- **sequences** – List of strings
- **concatenate** – Whether or not to concatenate the parsed strings. Defaults to False.

Returns
Dictionary containing parsed strings

Raises
ChemistryError – If the number sequences does not equal *n*

parse_reads(*reads: List[*ngs_tools.fastq.Read.Read*]*, *concatenate: bool = False*, *check_name: bool = True*) → Dict[str, Tuple[Union[str, Tuple[str]], Union[str, Tuple[str]]]]

Behaves identically to *parse()* but on a list of *ngs_tools.fastq.Read* instances. The resulting dictionary contains tuple values, where the first element corresponds to the parsed read sequences, while the second corresponds to the parsed quality strings.

Parameters

- **reads** – List of *ngs_tools.fastq.Read* instances
- **concatenate** – Whether or not to concatenate the parsed strings. Defaults to False.
- **check_name** – If True, raises *ChemistryError* if all the reads do not have the same name. Defaults to True.

Returns
Dictionary containing tuples of parsed read sequences and quality strings

Raises
ChemistryError – If the number sequences does not equal *n*, or `check_name=True` and not all reads have the same name.

__eq__(*other:* Chemistry)

Check the equality of two chemistries by comparing each parser.

__str__()

Return str(self).

__repr__()

Return repr(self).

to_kallisto_bus_arguments() → Dict[str, str]

Convert this spatial chemistry definition to arguments that can be used as input to kallisto bus. https://www.kallistobus.tools/

Returns
A Dictionary of arguments-to-value mappings. For this particular function, the dictionary has a single *-x* key and the value is a custom technology definition string, as specified in the kallisto manual.

to_starsolo_arguments() → Dict[str, str]

Converts this spatial chemistry definition to arguments that can be used as input to STARsolo. https://github.com/alexdobin/STAR/blob/master/docs/STARsolo.md

Returns
A Dictionary of arguments-to-value mappings.

ngs_tools.chemistry.MultimodalChemistry

Module Contents

Classes

MultimodalChemistry	Represents any chemistry that is a combination of multiple chemistries.

Attributes

_10X_FEATUREBARCODE
_10X_MULTIOME
MULTIMODAL_CHEMISTRIES

exception ngs_tools.chemistry.MultimodalChemistry.**MultimodalChemistryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.MultimodalChemistry.**MultimodalChemistry**(*name: str*, *description: str*, *chemistries: Dict[str,* ngs_tools.chemistry.Chemistry.Chemistry*]*)

Represents any chemistry that is a combination of multiple chemistries. For example, 10x Multiome. Note that this is not a subclass of Chemistry.

TODO: Add properties similar to Chemistry class.

property name: str

Chemistry name

property description: str

Chemistry description

property chemistries: Dict[str, *ngs_tools.chemistry.Chemistry.Chemistry*]

chemistry(*key: str*) → *ngs_tools.chemistry.Chemistry.Chemistry*

ngs_tools.chemistry.MultimodalChemistry.**_10X_FEATUREBARCODE**

ngs_tools.chemistry.MultimodalChemistry.**_10X_MULTIOME**

ngs_tools.chemistry.MultimodalChemistry.**MULTIMODAL_CHEMISTRIES**

ngs_tools.chemistry.SingleCellChemistry

Module Contents

Classes

SingleCellChemistry	Extends `SequencingChemistry` to be able to handle common single-cell

Attributes

_10X_V1
_10X_V2
_10X_V3
_10X_V3_ULTIMA
_10X_FB
_10X_ATAC
_DROPSEQ
_CELSEQ_V1
_CELSEQ_V2
_INDROPS_V1
_INDROPS_V2
_INDROPS_V3
_SCRBSEQ
_SURECELL
_SMARTSEQ_V2
_SMARTSEQ_V3
_STORMSEQ
_SCI_FATE
_BDWTA
_SPLITSEQ
_PLATE_SINGLE_CELL_CHEMISTRIES
_DROPLET_SINGLE_CELL_CHEMISTRIES
_OTHER_SINGLE_CELL_CHEMISTRIES
SINGLE_CELL_CHEMISTRIES

exception ngs_tools.chemistry.SingleCellChemistry.**SingleCellChemistryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.SingleCellChemistry.**SingleCellChemistry**(*name: str*, *description: str*, *n: int*, *strand: ngs_tools.chemistry.Chemistry.SequencingStrand*, *cdna_parser: ngs_tools.chemistry.Chemistry.SubSequencePars*, *cell_barcode_parser: Optional[ngs_tools.chemistry.Chemistry.SubSequer* = *None*, *umi_parser: Optional[ngs_tools.chemistry.Chemistry.SubSequer* = *None*, *whitelist_path: Optional[str]* = *None*)

Bases: *ngs_tools.chemistry.Chemistry.SequencingChemistry*

Extends SequencingChemistry to be able to handle common single-cell chemistries.

property cell_barcode_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the cell barcode parser

property barcode_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the cell barcode parser

property umi_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the UMI parser

property cdna_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the cDNA parser

property has_cell_barcode: **bool**

Whether the chemistry has a cell barcode

property has_barcode: **bool**

Whether the chemistry has a cell barcode

property has_umi: **bool**

Whether the chemistry has a UMI

property has_whitelist: **bool**

Whether the chemistry has a fixed predefined cell barcode whitelist

property whitelist_path: **str**

Path to the whitelist

ngs_tools.chemistry.SingleCellChemistry.**_10X_V1**

ngs_tools.chemistry.SingleCellChemistry.**_10X_V2**

ngs_tools.chemistry.SingleCellChemistry.**_10X_V3**

ngs_tools.chemistry.SingleCellChemistry.**_10X_V3_ULTIMA**

ngs_tools.chemistry.SingleCellChemistry.**_10X_FB**

ngs_tools.chemistry.SingleCellChemistry.**_10X_ATAC**

ngs_tools.chemistry.SingleCellChemistry.**_DROPSEQ**

ngs_tools.chemistry.SingleCellChemistry._CELSEQ_V1

ngs_tools.chemistry.SingleCellChemistry._CELSEQ_V2

ngs_tools.chemistry.SingleCellChemistry._INDROPS_V1

ngs_tools.chemistry.SingleCellChemistry._INDROPS_V2

ngs_tools.chemistry.SingleCellChemistry._INDROPS_V3

ngs_tools.chemistry.SingleCellChemistry._SCRBSEQ

ngs_tools.chemistry.SingleCellChemistry._SURECELL

ngs_tools.chemistry.SingleCellChemistry._SMARTSEQ_V2

ngs_tools.chemistry.SingleCellChemistry._SMARTSEQ_V3

ngs_tools.chemistry.SingleCellChemistry._STORMSEQ

ngs_tools.chemistry.SingleCellChemistry._SCI_FATE

ngs_tools.chemistry.SingleCellChemistry._BDWTA

ngs_tools.chemistry.SingleCellChemistry._SPLITSEQ

ngs_tools.chemistry.SingleCellChemistry._PLATE_SINGLE_CELL_CHEMISTRIES

ngs_tools.chemistry.SingleCellChemistry._DROPLET_SINGLE_CELL_CHEMISTRIES

ngs_tools.chemistry.SingleCellChemistry._OTHER_SINGLE_CELL_CHEMISTRIES

ngs_tools.chemistry.SingleCellChemistry.SINGLE_CELL_CHEMISTRIES

ngs_tools.chemistry.SpatialChemistry

Module Contents

Classes

SpatialResolution	Typed version of namedtuple.
SpatialChemistry	Extends `Chemistry` to be able to handle spatial chemistries.
SpatialSequencingChemistry	Extends `SequencingChemistry` to be able to handle common spatial chemistries.

Attributes

_SLIDESEQ_V2
_VISIUM
_STEREOSEQ
_COSMX
_SEQSCOPE_HDMI_DRAI
_SEQSCOPE_HDMI32_DRA1
_STARMAP
_SEQFISH
_MERFISH
_SEQUENCING_SPATIAL_CHEMISTRIES
_INSITU_SPATIAL_CHEMISTRIES
SPATIAL_CHEMISTRIES

class ngs_tools.chemistry.SpatialChemistry.**SpatialResolution**

Bases: NamedTuple

Typed version of namedtuple.

Usage in Python versions >= 3.6:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has extra __annotations__ and _field_types attributes, giving an ordered dict mapping field names to types. __annotations__ should be preferred, while _field_types is kept to maintain pre PEP 526 compatibility. (The field names are in the _fields attribute, which is part of the namedtuple API.) Alternative equivalent keyword syntax is also accepted:

```
Employee = NamedTuple('Employee', name=str, id=int)
```

In Python versions <= 3.5 use:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

scale: `float = 1.0`

unit: `Optional[typing_extensions.Literal[nm, um, mm]]`

exception ngs_tools.chemistry.SpatialChemistry.**SpatialChemistryError**

Bases: `Exception`

Common base class for all non-exit exceptions.

class ngs_tools.chemistry.SpatialChemistry.**SpatialChemistry**(*resolution:* SpatialResolution, **args*, ***kwargs*)

Bases: *ngs_tools.chemistry.Chemistry.Chemistry*

Extends `Chemistry` to be able to handle spatial chemistries.

property resolution: *SpatialResolution*

Get the spatial resolution as a *SpatialResolution* object.

class ngs_tools.chemistry.SpatialChemistry.**SpatialSequencingChemistry**(*name: str*, *description: str*, *resolution:* SpatialResolution, *n: int*, *strand:* ngs_tools.chemistry.Chemistry.SequencingS *cdna_parser:* ngs_tools.chemistry.Chemistry.SubSequenc *spot_barcode_parser: Optional[*ngs_tools.chemistry.Chemistry.SubS *= None*, *umi_parser: Optional[*ngs_tools.chemistry.Chemistry.SubS *= None*, *whitelist_path: Optional[str] = None*)

Bases: *SpatialChemistry*, *ngs_tools.chemistry.Chemistry.SequencingChemistry*

Extends `SequencingChemistry` to be able to handle common spatial chemistries.

property spot_barcode_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the spot barcode parser

property barcode_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the spot barcode parser

property umi_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the UMI parser

property cdna_parser: *ngs_tools.chemistry.Chemistry.SubSequenceParser*

Get the cDNA parser

property has_spot_barcode: `bool`

Whether the chemistry has a spot barcode

property has_barcode: `bool`

Whether the chemistry has a spot barcode

property has_umi: `bool`

Whether the chemistry has a UMI

property has_whitelist: bool

Whether the chemistry has a fixed predefined spot barcode whitelist

property whitelist_path: Optional[str]

Path to the whitelist. None if it does not exist.

ngs_tools.chemistry.SpatialChemistry.**_SLIDESEQ_V2**

ngs_tools.chemistry.SpatialChemistry.**_VISIUM**

ngs_tools.chemistry.SpatialChemistry.**_STEREOSEQ**

ngs_tools.chemistry.SpatialChemistry.**_COSMX**

ngs_tools.chemistry.SpatialChemistry.**_SEQSCOPE_HDMI_DRAI**

ngs_tools.chemistry.SpatialChemistry.**_SEQSCOPE_HDMI32_DRA1**

ngs_tools.chemistry.SpatialChemistry.**_STARMAP**

ngs_tools.chemistry.SpatialChemistry.**_SEQFISH**

ngs_tools.chemistry.SpatialChemistry.**_MERFISH**

ngs_tools.chemistry.SpatialChemistry.**_SEQUENCING_SPATIAL_CHEMISTRIES**

ngs_tools.chemistry.SpatialChemistry.**_INSITU_SPATIAL_CHEMISTRIES**

ngs_tools.chemistry.SpatialChemistry.**SPATIAL_CHEMISTRIES**

Package Contents

Functions

_clean_name(→ Tuple[str, Optional[int]])	Internal helper function to clean chemistry names.
get_chemistry(→ Chemistry.Chemistry)	Fetch a *Chemistry* definition by name. Uses some regex magic to

Attributes

VERSION_PARSER
CHEMISTRIES

ngs_tools.chemistry.**VERSION_PARSER**

ngs_tools.chemistry.**CHEMISTRIES**

ngs_tools.chemistry.**_clean_name**(*name: str*) → Tuple[str, Optional[int]]

Internal helper function to clean chemistry names.

Parameters
name – String name of the chemistry.

Returns
Tuple of the cleaned name and version

ngs_tools.chemistry.**get_chemistry**(*name: str*) → *Chemistry.Chemistry*

Fetch a *Chemistry* definition by name. Uses some regex magic to correctly deal with chemistry versioning at the end of the name. For instance, `10x2` is interpreted the same as `10xv2`.

See *SingleCellChemistry* and *SpatialChemistry* for available chemistries.

Parameters
name – String name of the chemistry. Any dashes (-) or capitalization are ignored.

Returns
The matching chemistry.

Raises
ChemistryError – If the chemistry could not be found.

1.1.3 ngs_tools.fasta

Submodules

ngs_tools.fasta.Fasta

Module Contents

Classes

Fasta	Represents a single FASTA file.

exception ngs_tools.fasta.Fasta.**FastaError**

Bases: `Exception`

Common base class for all non-exit exceptions.

class ngs_tools.fasta.Fasta.**Fasta**(*path: str*, *mode: typing_extensions.Literal[r, w] = 'r'*)

Bases: *ngs_tools.utils.FileWrapper*

Represents a single FASTA file.

_header

Variable that temporarily holds the header string for the next FASTA entry; for internal use only.

read() → *ngs_tools.fasta.FastaEntry.FastaEntry*

Read a single FASTA entry as a `FastaEntry` instance.

Returns
The next FASTA entry

Raises

- *FastaError* – If the file was not opened for reading, or the file was closed.
- **StopIteration** – When there are no more entries to read.

write(*entry:* ngs_tools.fasta.FastaEntry.FastaEntry)

Write a single FASTA entry.

Parameters
entry – The FASTA entry to write

Raises
FastaError – If the file was not opened for writing, or the file was closed.

ngs_tools.fasta.FastaEntry

Module Contents

Classes

FastaEntry	Represents a single FASTA entry, which consists of a header and a sequence.

exception ngs_tools.fasta.FastaEntry.**FastaEntryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.fasta.FastaEntry.**FastaEntry**(*header: str*, *sequence: str*)

Represents a single FASTA entry, which consists of a header and a sequence.

ATTRIBUTE_PARSER

Static attribute that is a compiled regex. Used to parse attributes.

_header

Header string; for internal use only. Use *header* instead.

_sequence

Sequence string; for internal use only. Use *sequence* instead.

property header: str

Header string, including the > character

property sequence: str

Sequence string

property name: str

Name of the sequence, which comes immedately after > in the header

property attributes: Dict[str, str]

Dictionary of entry attributes, parsed from the substring of the header after the first space character.

ATTRIBUTE_PARSER

static make_header(*name: str*, *attributes: Dict[str, str]*) → str

Static method to construct a header string from a name and attributes.

Parameters

- **name** – entry name
- **attributes** – dictionary containing entry attributes

Package Contents

Functions

split_genomic_fasta_to_cdna(→ str)	Split a genomic FASTA into cDNA by using gene and transcript information
split_genomic_fasta_to_intron(→ str)	Split a genomic FASTA into introns by using gene and transcript information
split_genomic_fasta_to_nascent(→ str)	Split a genomic FASTA into nascent transcripts by using gene information

ngs_tools.fasta.**split_genomic_fasta_to_cdna**(*fasta_path: str*, *out_path: str*, *gene_infos: dict*, *transcript_infos: dict*, *show_progress: bool = False*) → str

Split a genomic FASTA into cDNA by using gene and transcript information generated from extracting information from a GTF.

Parameters

- **fasta_path** – Path to FASTA containing genomic sequences
- **out_path** – Path to output FASTA that will contain cDNA sequences
- **gene_infos** – Dictionary containing gene information, as returned by *ngs_tools.gtf.genes_and_transcripts_from_gtf()*
- **transcript_infos** – Dictionary containing transcript information, as returned by *ngs_tools.gtf.genes_and_transcripts_from_gtf()*
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to written FASTA

ngs_tools.fasta.**split_genomic_fasta_to_intron**(*fasta_path: str*, *out_path: str*, *gene_infos: dict*, *transcript_infos: dict*, *flank: int = 30*, *show_progress: bool = False*) → str

Split a genomic FASTA into introns by using gene and transcript information generated from extracting information from a GTF. Optionally append flanking sequences and collapse introns that have overlapping flanking regions.

Parameters

- **fasta_path** – Path to FASTA containing genomic sequences
- **out_path** – Path to output FASTA that will contain cDNA sequences
- **gene_infos** – Dictionary containing gene information, as returned by *ngs_tools.gtf.genes_and_transcripts_from_gtf()*

- **transcript_infos** – Dictionary containing transcript information, as returned by *ngs_tools.gtf.genes_and_transcripts_from_gtf()*
- **flank** – Number of flanking bases to include for each intron. Defaults to 30.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns
Path to written FASTA

ngs_tools.fasta.**split_genomic_fasta_to_nascent**(*fasta_path: str*, *out_path: str*, *gene_infos: dict*, *suffix=''*, *show_progress: bool = False*) → str

Split a genomic FASTA into nascent transcripts by using gene information generated from extracting information from a GTF.

Parameters

- **fasta_path** – Path to FASTA containing genomic sequences
- **out_path** – Path to output FASTA that will contain cDNA sequences
- **gene_infos** – Dictionary containing gene information, as returned by *ngs_tools.gtf.genes_and_transcripts_from_gtf()*
- **suffix** – Suffix to append to output FASTA entry names. Defaults to "".
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns
Path to written FASTA

1.1.4 ngs_tools.fastq

Submodules

ngs_tools.fastq.Fastq

Module Contents

Classes

Fastq	Class that represents a FASTQ file.

exception ngs_tools.fastq.Fastq.**FastqError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.fastq.Fastq.**Fastq**(*path: str*, *mode: typing_extensions.Literal[r, w] = 'r'*)

Bases: *ngs_tools.utils.FileWrapper*

Class that represents a FASTQ file.

read() → *ngs_tools.fastq.Read.Read*

Read a single FASTQ entry.

Returns
The next read as a Read instance

Raises

- ***FastqError*** – If the file was not opened for reading, or the file was closed.
- **StopIteration** – When there are no more entries to read.

write(*entry:* ngs_tools.fastq.Read.Read)

Write a single FASTQ entry.

Parameters
entry – Read instance to write to the FASTQ

Raises
FastaError – If the file was not opened for writing, or the file was closed.

ngs_tools.fastq.Read

Module Contents

Classes

Quality	Represents a Phred33 quality string.
Read	Class that represents a FASTQ read. Once the class is initialized, with the

exception ngs_tools.fastq.Read.**ReadError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.fastq.Read.**Quality**(*qualities: str*)

Represents a Phred33 quality string.

_string

Raw quality string; for internal use only. Use *string* instead.

property string: str

Raw quality string

property values: str

List of quality values

property probs: str

The quality values converted to probabilities of error

__getitem__(*sl: slice*) → *Quality*

Return a slice of the *Quality*

class ngs_tools.fastq.Read.**Read**(*header: str*, *sequence: str*, *qualities: str*)

Class that represents a FASTQ read. Once the class is initialized, with the header, sequence, and quality strings, they should not be changed (hence using @property). Only Phred33 quality scores are supported, but there is no check for this.

_header

Raw header string, including the @; for internal use only. Use *header* instead.

_sequence

Raw sequence string; for internal use only. Use *sequence* instead.

_qualities

Quality instance representing the sequence qualities; for internal use only. Use *qualities* instead.

property header: str

Raw header string

property sequence: str

Raw sequence string

property qualities: str

Quality instance representing the sequence qualities

property name: str

Name of the sequence, which comes immediately after the `@`

property attributes: str

String of read attributes. This is the substring after the first space in the header.

Package Contents

Functions

fastq_to_bam(→ str)	Convert a Fastq to unmapped BAM.
fastqs_to_bam(→ str)	Convert FASTQs to an unmapped BAM according to an arbitrary function.
fastqs_to_bam_with_chemistry(→ str)	Convert FASTQs to an unmapped BAM according to the provided

ngs_tools.fastq.**fastq_to_bam**(*fastq_path: str*, *bam_path: str*, *name: Optional[str] = None*, *n_threads: int = 1*, *show_progress: bool = False*) → str

Convert a Fastq to unmapped BAM.

Parameters

- **fastq_path** – Path to the input FASTQ
- **bam_path** – Path to the output BAM
- **name** – Name for this set of reads. Defaults to None. If not provided, a random string is generated by calling `shortuuid.uuid()`. This value is added as the read group (RG tag) for all the reads in the BAM.
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to BAM

ngs_tools.fastq.**fastqs_to_bam**(*fastq_paths: List[str]*, *parse_func: Callable[[Tuple[*Read.Read*, *Ellipsis], pysam.AlignmentHeader], pysam.AlignedSegment]*, *bam_path: str*, *name: Optional[str] = None*, *n_threads: int = 1*, *show_progress: bool = False*) → str

Convert FASTQs to an unmapped BAM according to an arbitrary function.

Parameters

- **fastq_paths** – List of FASTQ paths.
- **parse_func** – Function that accepts a tuple of *ngs_tools.fastq.Read* objects (one from each FASTQ) and a `pysam.AlignmentHeader` object as the second argument, and returns a new `pysam.AlignedSegment` object to write into the BAM. Note that the second argument must be used for the *header* argument when initializing the new `pysam.AlignedSegment`. Whenever this function returns *None*, the read will not be written to the BAM.
- **name** – Name for this set of reads. Defaults to None. If not provided, a random string is generated by calling `shortuuid.uuid()`. This value is added as the read group (RG tag) for all the reads in the BAM.
- **bam_path** – Path to the output BAM
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to BAM

ngs_tools.fastq.**fastqs_to_bam_with_chemistry**(*fastq_paths: List[str]*, *chemistry: ngs_tools.chemistry.Chemistry*, *tag_map: Dict[str, Tuple[str, str]]*, *bam_path: str*, *name: Optional[str] = None*, *sequence_key: str = 'cdna'*, *n_threads: int = 1*, *show_progress: bool = False*) → str

Convert FASTQs to an unmapped BAM according to the provided *ngs_tools.chemistry.Chemistry* instance.

Note that any split features (i.e. split barcode where barcode is in multiple positions) are concatenated.

Parameters

- **fastq_paths** – List of FASTQ paths. The order must match that of the chemistry.
- **chemistry** – *ngs_tools.chemistry.Chemistry* instance to use to parse the reads.
- **tag_map** – Mapping of parser names to their corresponding BAM tags. The keys are the parser names, and the values must be a tuple of `(sequence BAM tag, quality BAM tag)`, where the former is the tag that will be used for the nucleotide sequence, and the latter is the tag that will be used for the quality scores.
- **bam_path** – Path to the output BAM
- **name** – Name for this set of reads. Defaults to None. If not provided, a random string is generated by calling `shortuuid.uuid()`. This value is added as the read group (RG tag) for all the reads in the BAM.
- **sequence_key** – Parser key to use as the actual alignment sequence. Defaults to *cdna*.
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to BAM

Raises

FastqError – If the number of FASTQs provided does not meet the number required for the specified chemistry, if the tag map provides keys that do not exist for the chemistry, or if the tag map contains multiple BAM tags.

1.1.5 ngs_tools.gtf

Submodules

ngs_tools.gtf.Gtf

Module Contents

Classes

Gtf	Class that represents a GTF file.

exception ngs_tools.gtf.Gtf.**GtfError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.gtf.Gtf.**Gtf**(*path: str*, *mode: typing_extensions.Literal[r, w] = 'r'*)

Bases: *ngs_tools.utils.FileWrapper*

Class that represents a GTF file.

read() → *ngs_tools.gtf.GtfEntry.GtfEntry*

Read a single GTF entry as a GtfEntry instance.

Returns
The next GTF entry

Raises

- ***GtfError*** – If the file was not opened for reading, or the file was closed.
- **StopIteration** – When there are no more entries to read.

write(*entry:* ngs_tools.gtf.GtfEntry.GtfEntry)

Write a single GTF entry.

Parameters
entry – The GTF entry to write

Raises
FastaError – If the file was not opened for writing, or the file was closed.

ngs_tools.gtf.GtfEntry

Module Contents

Classes

GtfEntry	Represents a single GTF entry.

exception ngs_tools.gtf.GtfEntry.**GtfEntryError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.gtf.GtfEntry.**GtfEntry**(*line: str*)

Represents a single GTF entry.

PARSER

Static attribute that contains a compiled regex. Used to parse a GTF line.

ATTRIBUTE_PARSER

Static attribute that contains a compiled regex. Used to parse GTF entry attributes.

_line

Raw GTF line; for internal use only. Use *line* instead.

_chromosome

Chromosome; for internal use only. Use *chromosome* instead.

_feature

Feature; for internal use only. Use *feature* instead.

_start

Start; for internal use only. Use *start* instead.

_end

End; for internal use only. Use *end* instead.

_strand

Strand; for internal use only. Use *strand* instead.

_attribute_str

Raw GTF entry attribute string; for internal use only. Use *attributes* instead.

property line: str

Raw GTF line

property chromosome: str

Chromosome

property feature: str

Feature

property start: int

Start, 1-indexed

property end: int

End, 1-indexed, inclusive

property strand: str

Strand

property attributes: Dict[str, str]

Dictionary of attributes

PARSER

ATTRIBUTE_PARSER

to_segment() → *ngs_tools.gtf.Segment.Segment*

Convert this GTF entry into a Segment.

Returns
The new segment

ngs_tools.gtf.Segment

Module Contents

Classes

Segment	Class to represent an integer interval segment, zero-indexed, start-inclusive

exception ngs_tools.gtf.Segment.**SegmentError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.gtf.Segment.**Segment**(*start: int*, *end: int*)

Class to represent an integer interval segment, zero-indexed, start-inclusive and end-exclusive.

_start

Segment start; for internal use only. Use *start* instead.

_end

Segment end; for internal use only. Use *end* instead.

property start: int

Segment start

property end: int

Segment end

property width: int

Segment width

is_in(*i: int*) → bool

Evaluate whether an integer is contained within the segment.

Parameters
i – Integer number to check

Returns
True or False

is_exclusive(*segment:* Segment) → bool

Evaluate whether this segment is exclusive of another segment.

Parameters
segment – *Segment* object to check

Returns
True or False

is_overlapping(*segment:* Segment) → bool

Evaluate whether this segment overlaps with another segment.

Parameters
segment – *Segment* object to check

Returns
True or False

is_subset(*segment:* Segment) → bool

Evaluate whether this segment is a subset of another segment.

Parameters
segment – *Segment* object to check

Returns
True or False

is_superset(*segment:* Segment) → bool

Evaluate whether this segment is a superset of another segment.

Parameters
segment – *Segment* object to check

Returns
True or False

flank(*l: int*, *left: Optional[int] = None*, *right: Optional[int] = None*) → *Segment*

Construct a new segment with start and end flanks of length `l`. Optionally, limit the size of the new segment.

Parameters

- **l** – Flank length
- **left** – Clip the resulting segment's start to this value. Defaults to None. If not provided, no clipping is performed.
- **right** – Clip the resulting segment's end to this value. Defaults to None. If not provided, no clipping is performed.

Returns
The new segment

__iter__()

__eq__(*other*)

Return self==value.

__lt__(*other*)

Return self<value.

__gt__(*other*)

Return self>value.

__repr__()

Return repr(self).

ngs_tools.gtf.SegmentCollection

Module Contents

Classes

SegmentCollection	Class to represent a collection of integer interval segments, zero-indexed.

exception ngs_tools.gtf.SegmentCollection.**SegmentCollectionError**

Bases: Exception

Common base class for all non-exit exceptions.

class ngs_tools.gtf.SegmentCollection.**SegmentCollection**(*segments: Optional[List[*ngs_tools.gtf.Segment.Segment*]] = None*)

Class to represent a collection of integer interval segments, zero-indexed. The segments are always sorted and overlaps are collapsed.

_segments

List of Segment instances; for internal use only. Use *segments* instead.

property segments: List[*ngs_tools.gtf.Segment.Segment*]

Get a list of Segment instances

property start: int

Leftmost value of all segments. 0 if there are no segments.

property end: int

Rightmost value of all segments. 0 if there are no segments.

add_segment(*segment:* ngs_tools.gtf.Segment.Segment)

Add a segment to the collection.

Parameters
segment – Segment to add

add_collection(*collection:* SegmentCollection)

Add a collection of segments to the collection.

Parameters
collection – Collection to add

__iter__()

__getitem__(*i*)

__len__()

__bool__()

__eq__(*other:* SegmentCollection)

Return self==value.

invert(*bounds:* ngs_tools.gtf.Segment.Segment) → *SegmentCollection*

Invert this *SegmentCollection* within the given bounds.

Parameters
bounds – The bounds to invert with respect to.

Returns
A new *SegmentCollection* that is inverted

Raises
SegmentCollectionError – If bounds does not entirely contain this collection

collapse()

Collapse the segments in this collection such that there are no overlapping segments. Any overlapping segments are merged into a single large segment.

span_is_exclusive(*collection:* SegmentCollection) → bool

Evaluate whether the span of this collection is exclusive of that of another collection.

Parameters
collection – *SegmentCollection* object to check

Returns
True or False

is_overlapping(*collection:* SegmentCollection) → bool

Evaluate whether this collection overlaps with another collection.

Parameters
collection – *SegmentCollection* object to check

Returns
True or False

is_subset(*collection:* SegmentCollection) → bool

Evaluate whether this collection is a subset of another collection.

Parameters
collection – *SegmentCollection* object to check

Returns
True or False

is_superset(*collection:* SegmentCollection) → bool

Evaluate whether this collection is a superset of another collection.

Parameters
collection – *SegmentCollection* object to check

Returns
True or False

flank(*l: int*, *left: Optional[int] = None*, *right: Optional[int] = None*) → *SegmentCollection*

Construct a new segment collection where all the segments have start and end flanks of length `l`. Optionally, limit the span of the new segment collection. This is done by calling `Segment.flank()` on all the segments and initializing a new *SegmentCollection*. Any overlaps are collapsed.

Parameters

- **l** – Flank length

- **left** – Clip the resulting collection's start to this value. Defaults to None. If not provided, no clipping is performed.
- **right** – Clip the resulting collection's end to this value. Defaults to None. If not provided, no clipping is performed.

Returns
The new collection

classmethod from_positions(*positions: Union[List[int], Set[int]]*) → *SegmentCollection*

Initialize a new collection given a list or set of integer positions.

Parameters
positions – Integer positions to construct the collection from

Returns
The new collection

classmethod from_collections(**collections:* SegmentCollection) → *SegmentCollection*

Initialize a new collection given an arbitrary number of collections.

Parameters
***collections** – The collections to combine

Returns
The new collection

__repr__()

Return repr(self).

Package Contents

Functions

parse_gtf(→ Generator[GtfEntry.GtfEntry, None, None])	Parse GTF and yield only the specified features as *GtfEntry* instances.
genes_and_transcripts_from_gtf(→ Tuple[dict, dict])	Parse GTF for gene and transcript information. Also, compute the introns of

ngs_tools.gtf.**parse_gtf**(*gtf_path: str*, *filter_func: Callable[[*GtfEntry.GtfEntry*], bool] = lambda entry: ...*, *show_progress: bool = False*) → Generator[*GtfEntry.GtfEntry*, None, None]

Parse GTF and yield only the specified features as *GtfEntry* instances.

Parameters

- **gtf_path** – path to GTF file
- **filter_func** – Function that takes a *GtfEntry* instance and returns `True` for entries to process and `False` for entries to ignore. Defaults to no filtering.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Yields
GTF entries

ngs_tools.gtf.**genes_and_transcripts_from_gtf**(*gtf_path: str*, *use_version: bool = False*, *filter_func: Callable[[*GtfEntry.GtfEntry*], bool] = lambda entry: ...*, *show_progress: bool = False*) → Tuple[dict, dict]

Parse GTF for gene and transcript information. Also, compute the introns of each transcript.

Parameters

- **gtf_path** – path to GTF file
- **use_version** – whether or not to use gene and transcript versions
- **filter_func** – Function that takes a *GtfEntry* instance and returns `True` for entries to process and `False` for entries to ignore. Defaults to no filtering.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Dictionary containing gene information Dictionary containing transcript information

1.2 Submodules

1.2.1 ngs_tools.bam

Module Contents

Functions

map_bam(bam_path, map_func[, n_threads, show_progress])		Generator to map an arbitrary function to every read and return its return
apply_bam(bam_path, apply_func, out_path[, n_threads, ...])		Apply an arbitrary function to every read in a BAM. Reads for which the
count_bam(→ int)		Count the number of BAM entries. Optionally, a function may be provided to
split_bam(→ Dict[str, Tuple[str, int]])		Split a BAM into many parts, either by the number of reads or by an
tag_bam_with_fastq(bam_path, fastq_path, tag_func, ...)		Add tags to BAM entries using sequences from one or more FASTQ files.
filter_bam(bam_path, filter_func, out_path[, ...])		Filter a BAM by applying the given function to each `pysam.AlignedSegment`

exception ngs_tools.bam.**BamError**

Bases: `Exception`

Common base class for all non-exit exceptions.

ngs_tools.bam.**map_bam**(*bam_path: str*, *map_func: Callable[[pysam.AlignedSegment], Any]*, *n_threads: int = 1*, *show_progress: bool = False*)

Generator to map an arbitrary function to every read and return its return values.

Parameters

- **bam_path** – Path to the BAM file
- **map_func** – Function that takes a `pysam.AlignedSegment` object and returns some value
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Yields
map_func applied to each read in the BAM file

ngs_tools.bam.**apply_bam**(*bam_path: str*, *apply_func: Callable[[pysam.AlignedSegment], Optional[pysam.AlignedSegment]]*, *out_path: str*, *n_threads: int = 1*, *show_progress: bool = False*)

Apply an arbitrary function to every read in a BAM. Reads for which the function returns *None* are not written to the output BAM.

Parameters

- **bam_path** – Path to the BAM file
- **apply_func** – Function that takes a `pysam.AlignedSegment` object and optionally returns `pysam.AlignedSegment` objects
- **out_path** – Path to output BAM file
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns
Path to written BAM

ngs_tools.bam.**count_bam**(*bam_path: str*, *filter_func: Optional[Callable[[pysam.AlignedSegment], bool]] = None*, *n_threads: int = 1*, *show_progress: bool = False*) → int

Count the number of BAM entries. Optionally, a function may be provided to only count certain alignments.

Parameters

- **bam_path** – Path to BAM
- **filter_func** – Function that takes a `pysam.AlignedSegment` object and returns True for reads to be counted and False otherwise
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns
Number of alignments in BAM

ngs_tools.bam.**split_bam**(*bam_path: str*, *split_prefix: str*, *split_func: Optional[Callable[[pysam.AlignedSegment], str]] = None*, *n: Optional[int] = None*, *n_threads: int = 1*, *check_pair_groups: bool = True*, *show_progress: bool = False*) → Dict[str, Tuple[str, int]]

Split a BAM into many parts, either by the number of reads or by an arbitrary function. Only one of `split_func` or `n` must be provided. Read pairs are always written to the same file.

This function makes two passes through the BAM file. The first pass is to identify which reads must be written together (i.e. are pairs). The second pass is to actually extract the reads and write them to the appropriate split.

The following procedure is used to identify pairs. 1) The `.is_paired` property is checked to be True. 2) If the read is uanligned, at most one other unaligned read with the same

read name is allowed to be in the BAM. This other read is its mate. If the read is aligned, it should have the `HI` BAM tag indicating the alignment index. If no `HI` tag is present, then it is assumed only one alignment should be present for each read pair. If any of these constraints are not met, an exception is raised.

Parameters

- **bam_path** – Path to the BAM file
- **split_prefix** – File path prefix to all the split BAMs
- **split_func** – Function that takes a `pysam.AlignedSegment` object and returns a string ID that is used to group reads into splits. All reads with a given ID will be written to a single BAM. Defaults to None.
- **n** – Number of BAMs to split into. Defaults to None.
- **n_threads** – Number of threads to use. Only affects reading. Writing is still serialized. Defaults to 1.
- **check_pair_groups** – When using `split_func`, make sure that paired reads are assigned the same ID (and thus are split into the same BAM). Defaults to True.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Dictionary of tuples, where the first element is the path to a split BAM, and the second element is the number of BAM entries written to that split. The keys are either the string ID of each split (if `split_func` is used) or the split index (if `n` is used), and the values are paths.

Raises

BamError – If any pair constraints are not met.

ngs_tools.bam.**tag_bam_with_fastq**(*bam_path: str*, *fastq_path: Union[str, List[str]]*, *tag_func: Union[Callable[[ngs_tools.fastq.Read], dict], List[Callable[[ngs_tools.fastq.Read], dict]]]*, *out_path: str*, *check_name: bool = True*, *n_threads: int = 1*, *show_progress: bool = False*)

Add tags to BAM entries using sequences from one or more FASTQ files.

Internally, this function calls *apply_bam()*.

Note: The tag keys generated from *tag_func* must contain unique keys of at most 2 characters.

Parameters

- **bam_path** – Path to the BAM file
- **fastq_path** – Path to FASTQ file. This option may be a list to extract tags from multiple FASTQ files. In this case, *tag_func* must also be a list of functions.
- **tag_func** – Function that takes a *ngs_tools.fastq.Read* object and returns a dictionary of tags. When multiple FASTQs are being parsed simultaneously, each function needs to produce unique dictionary keys. Additionally, BAM tag keys may only be at most 2 characters. However, neither of these conditions are checked in favor of runtime.
- **out_path** – Path to output BAM file
- **check_name** – Whether or not to raise a *BamError* if the FASTQ does not contain a read in the BAM
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to written BAM

Raises

BamError – If only one of *fastq_path* and *tag_func* is a list, if both are lists but they have different lengths, if *check_name=True* but there are missing tags.

ngs_tools.bam.**filter_bam**(*bam_path: str*, *filter_func: Callable[[pysam.AlignedSegment], bool]*, *out_path: str*, *n_threads: int = 1*, *show_progress: bool = False*)

Filter a BAM by applying the given function to each `pysam.AlignedSegment` object. When the function returns False, the read is not written to the output BAM.

Internally, this function calls *apply_bam()*.

Parameters

- **bam_path** – Path to the BAM file
- **filter_func** – Function that takes a `pysam.AlignedSegment` object and returns False for reads to be filtered out
- **out_path** – Path to output BAM file
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to False.

Returns

Path to written BAM

1.2.2 ngs_tools.logging

Module Contents

Classes

Logger	Custom logger that provides namespacing functionality.

Functions

silence_logger(name)	Given a logger name, silence it completely.
set_logger(log)	Set the logger to the provided *Logger* instance. Use this function

Attributes

logger

ngs_tools.logging.**silence_logger**(*name: str*)

Given a logger name, silence it completely.

Parameters

name – Logger name to silence

class ngs_tools.logging.**Logger**(*name: Optional[str] = None*)

Custom logger that provides namespacing functionality.

The following loggers are silenced by default:
anndata, h5py, numba, pysam, pystan

FORMAT

Static attribute that encodes the logging format string

logger

The logger object

ch

A `logging.Streamhandler` object that sets the format

FORMAT = '[%(asctime)s] %(levelname)7s %(message)s'

namespaced(*namespace: str*)

Function decorator to set the logging namespace for the duration of the function.

Parameters
namespace – The namespace

namespaced_context(*namespace: str*)

Context manager to set the logging namespace.

Parameters
namespace – The namespace

namespace_message(*message: str*) → str

Add namespace information at the beginning of the logging message.

Parameters
message – The logging message

Returns
The namespaced message

addHandler(*hdlr: logging.Handler*, *format: bool = True*)

removeHandler(**args*, ***kwargs*)

setLevel(**args*, ***kwargs*)

debug(*message*, **args*, ***kwargs*)

info(*message*, **args*, ***kwargs*)

warning(*message*, **args*, ***kwargs*)

exception(*message*, **args*, ***kwargs*)

critical(*message*, **args*, ***kwargs*)

error(*message*, **args*, ***kwargs*)

ngs_tools.logging.**logger**

ngs_tools.logging.**set_logger**(*log:* Logger)

Set the logger to the provided *Logger* instance. Use this function to override the default logger for this (ngs-tools) library from libraries that use this library (ngs-tools) as a dependency.

Parameters
log – The logger

1.2.3 ngs_tools.progress

Module Contents

ngs_tools.progress.**progress**

1.2.4 ngs_tools.sequence

Module Contents

Functions

alignment_to_cigar(→ str)	Convert an alignment to a CIGAR string.
complement_sequence(→ str)	Complement the given sequence, with optional reversing.
_sequence_to_array(→ numpy.ndarray)	
_qualities_to_array(→ numpy.ndarray)	
_most_likely_array(→ numpy.ndarray)	
_most_likely_sequence(→ str)	
_disambiguate_sequence(→ List[str])	
_calculate_positional_probs(→ numpy.ndarray)	
call_consensus(sequences[, proportion])	Call consensus sequences from a set of sequences. Internally, this
call_consensus_with_qualities(→ Union[Tuple[List[str], ...)	Given a list of sequences and their base qualities, constructs a *set* of consensus
levenshtein_distance(→ int)	Calculate the Levenshtein (edit) distance between two sequences.
_mismatch_mask(→ int)	
_mismatch_masks(→ numpy.ndarray)	
_hamming_distance(→ int)	
hamming_distance(→ int)	Calculate the hamming distance between two sequences.
_hamming_distances(→ numpy.ndarray)	
hamming_distances(→ numpy.ndarray)	Calculate the hamming distance between a sequence and a list of sequences.
_hamming_distance_matrix(→ numpy.ndarray)	
hamming_distance_matrix(→ numpy.ndarray)	Calculate all pairwise hamming distances between two lists of sequences.
_pairwise_hamming_distances(→ numpy.ndarray)	
pairwise_hamming_distances(→ numpy.ndarray)	Calculate all pairwise hamming distances between combinations of sequences
_correct_to_whitelist(→ Tuple[int, float])	
correct_sequences_to_whitelist(→ List[Union[str, None]])	Correct a list of sequences to a whitelist within *d* hamming distance.
correct_sequences_to_whitelist_simple(→ Dict[str, ...)	Correct a list of sequences to a whitelist within *d* hamming distance.

Attributes

NeedlemanWunsch
NUCLEOTIDES_STRICT
NUCLEOTIDES_PERMISSIVE
NUCLEOTIDES
NUCLEOTIDES_AMBIGUOUS
NUCLEOTIDE_COMPLEMENT
NUCLEOTIDE_MASKS
MASK_TO_NUCLEOTIDE
LEVENSHTEIN_DISTANCE_ALIGNER
SEQUENCE_PARSER

ngs_tools.sequence.**NeedlemanWunsch**

ngs_tools.sequence.**NUCLEOTIDES_STRICT = ['A', 'C', 'G', 'T']**

ngs_tools.sequence.**NUCLEOTIDES_PERMISSIVE = ['R', 'Y', 'S', 'W', 'K', 'M', 'B', 'D', 'H', 'V', 'N', '-']**

ngs_tools.sequence.**NUCLEOTIDES**

ngs_tools.sequence.**NUCLEOTIDES_AMBIGUOUS**

ngs_tools.sequence.**NUCLEOTIDE_COMPLEMENT**

ngs_tools.sequence.**NUCLEOTIDE_MASKS**

ngs_tools.sequence.**MASK_TO_NUCLEOTIDE**

ngs_tools.sequence.**LEVENSHTEIN_DISTANCE_ALIGNER**

ngs_tools.sequence.**SEQUENCE_PARSER**

exception ngs_tools.sequence.**SequenceError**

Bases: Exception

Common base class for all non-exit exceptions.

ngs_tools.sequence.**alignment_to_cigar**(*reference: str*, *query: str*, *mismatch: bool = False*) → str

Convert an alignment to a CIGAR string.

The CIGAR is always constructed relative to the *reference* (i.e. as insertions/deletions from the reference). The provided sequences must represent their alignments, containing the "-" character at positions where there are gaps in one or another.

Parameters

- **reference** – The reference sequence alignment
- **query** – The query sequence alignment
- **mismatch** – Whether or not to use the "X" CIGAR operation in places of mismatches. Defaults to *True*, such that all non-gaps are considered matches with the "M" CIGAR operation. When there are ambiguous characters, a mismatch occurs only when the two sets of possible nucleotides are exclusive.

Returns
The CIGAR string representing the provided alignment

Raises
SequenceError – if the *reference* and *query* do not have the same length, or if there are gaps in both alignments at the same position

ngs_tools.sequence.**complement_sequence**(*sequence: str*, *reverse: bool = False*) → str

Complement the given sequence, with optional reversing.

Parameters

- **sequence** – Input sequence
- **reverse** – Whether or not to perform reverse complementation

Returns
Complemented (and optionally reversed) string

ngs_tools.sequence.**_sequence_to_array**(*sequence: str*, *l: Optional[int] = None*) → numpy.ndarray

ngs_tools.sequence.**_qualities_to_array**(*qualities: Union[str, array.array]*, *l: Optional[int] = None*) → numpy.ndarray

ngs_tools.sequence.**_most_likely_array**(*positional_probs: numpy.ndarray*) → numpy.ndarray

ngs_tools.sequence.**_most_likely_sequence**(*positional_probs: numpy.ndarray*, *allow_ambiguous: bool = False*) → str

ngs_tools.sequence.**_disambiguate_sequence**(*sequence: numpy.ndarray*) → List[str]

ngs_tools.sequence.**_calculate_positional_probs**(*sequences: numpy.ndarray*, *qualities: numpy.ndarray*) → numpy.ndarray

ngs_tools.sequence.**call_consensus**(*sequences: List[str]*, *proportion: float = 0.05*)

Call consensus sequences from a set of sequences. Internally, this function calls *call_consensus_with_qualities()* with all qualities set to 1 and `q_threshold` set to 1. See the documentation of this function for details on how consensuses are called.

Parameters

- **sequences** – Sequences to call consensus for
- **proportion** – Proportion of each sequence to allow mismatched bases to be above `q_threshold`

Returns
List of consensus sequences Numpy array of assignments for each sequence in `sequences`

ngs_tools.sequence.**call_consensus_with_qualities**(*sequences: List[str]*, *qualities: Union[List[str], List[array.array], List[numpy.ndarray]]*, *q_threshold: Optional[int] = None*, *proportion: float = 0.05*, *allow_ambiguous: bool = False*, *return_qualities: bool = False*) → Union[Tuple[List[str], numpy.ndarray], Tuple[List[str], numpy.ndarray, List[str]]]

Given a list of sequences and their base qualities, constructs a *set* of consensus sequences by iteratively constructing a consensus (by selecting the most likely base at each position) and assigning sequences with match probability <= max(min(match probability), *q_threshold* * (*proportion* * length of longest sequence)) to this consensus. Then, the consensus is updated by constructing the consensus only among these sequences. The match probability of a sequence to a consensus is the sum of the quality values where they do not match (equivalent to negative log probability that all mismatches were sequencing errors).

Note: This function does not perform any alignment among consensus sequences. To detect any insertions/deletions, call this function and then perform alignment among the called consensus sequences.

Parameters

- **sequences** – Sequences to call consensus for
- **qualities** – Quality scores for the sequences
- **q_threshold** – Quality threshold. Defaults to the median quality score among all bases in all sequences.
- **proportion** – Proportion of each sequence to allow mismatched bases to be above `q_threshold`. Defaults to 0.05.
- **allow_ambiguous** – Allow ambiguous bases in the consensus sequences. Defaults to False, which, on ties, selects a single base lexicographically. This option only has an effect when constructing the final consensus sequence as a string, not when calculating error probabilities.
- **return_qualities** – Whether or not to return qualities for the consensuses. Defaults to False.

Returns

List of consensus sequences Numpy array of assignments for each sequence in `sequences` Qualities for each of the consensus sequences, if `return_qualities` is True

Raises

SequenceError – if any sequence-quality pair have different lengths or number of provided sequences does not match number of provided qualities

ngs_tools.sequence.**levenshtein_distance**(*sequence1: str*, *sequence2: str*) → int

Calculate the Levenshtein (edit) distance between two sequences. This is calculated by calling `pyseq_align.NeedlemanWunsch.align()` with the appropriate scores/penalties.

Parameters

- **sequence1** – First sequence
- **sequence2** – Second sequence

Returns

Levenshtein distance

ngs_tools.sequence.**_mismatch_mask**(*sequence1: numpy.ndarray*, *sequence2: numpy.ndarray*) → int

ngs_tools.sequence.**_mismatch_masks**(*sequence: numpy.ndarray*, *whitelist: numpy.ndarray*, *d: int*) → numpy.ndarray

ngs_tools.sequence.**_hamming_distance**(*sequence1: numpy.ndarray*, *sequence2: numpy.ndarray*) → int

ngs_tools.sequence.**hamming_distance**(*sequence1: str*, *sequence2: str*) → int

Calculate the hamming distance between two sequences.

Parameters

- **sequence1** – First sequence
- **sequence2** – Second sequence

Returns

Hamming distance

Raises

SequenceError – When the sequences are unequal lengths

ngs_tools.sequence.**_hamming_distances**(*sequence: numpy.ndarray*, *sequences: numpy.ndarray*) → numpy.ndarray

ngs_tools.sequence.**hamming_distances**(*sequence: str*, *sequences: List[str]*) → numpy.ndarray

Calculate the hamming distance between a sequence and a list of sequences.

Parameters

- **sequence** – Sequence
- **sequences** – List of sequences

Returns

Numpy array of hamming distances, where each index `i` corresponds to the hamming distance between `sequence` and `sequences[i]`

Raises

SequenceError – When any of the sequences are unequal length

ngs_tools.sequence.**_hamming_distance_matrix**(*sequences1: numpy.ndarray*, *sequences2: numpy.ndarray*) → numpy.ndarray

ngs_tools.sequence.**hamming_distance_matrix**(*sequences1: List[str]*, *sequences2: List[str]*) → numpy.ndarray

Calculate all pairwise hamming distances between two lists of sequences.

Parameters

- **sequences1** – List of sequences
- **sequences2** – List of sequences

Returns

Numpy array of hamming distances, where each index `(i, j)` corresponds to the hamming distance between `sequences1[i]` and `sequences2[j]`

Raises

SequenceError – When any of the sequences are unequal length

ngs_tools.sequence.**_pairwise_hamming_distances**(*sequences: numpy.ndarray*) → numpy.ndarray

ngs_tools.sequence.**pairwise_hamming_distances**(*sequences: List[str]*) → numpy.ndarray

Calculate all pairwise hamming distances between combinations of sequences from a single list.

Parameters
sequences – List of sequences

Returns
Numpy array of hamming distances, where each index `(i, j)` corresponds to the hamming distance between `sequences[i]` and `sequences[j]`

Raises
SequenceError – When any of the sequences are unequal length

ngs_tools.sequence.**_correct_to_whitelist**(*qualities: numpy.ndarray*, *indices: numpy.ndarray*, *masks: numpy.ndarray*, *log10_proportions: numpy.ndarray*) → Tuple[int, float]

ngs_tools.sequence.**correct_sequences_to_whitelist**(*sequences: List[str]*, *qualities: Union[List[str], List[array.array]]*, *whitelist: List[str]*, *d: int = 1*, *confidence: float = 0.95*, *n_threads: int = 1*, *show_progress: bool = False*) → List[Union[str, None]]

Correct a list of sequences to a whitelist within *d* hamming distance. Note that *sequences* can contain duplicates, but *whitelist* can not.

For a given sequence, if there are multiple barcodes in the whitelist to which its distance is <= *d*, the sequence is assigned to a barcode by using the prior probability that the sequence originated from the barcode. If the confidence that the sequence originated from the most likely barcode is less than *confidence*, assignment is skipped.

This procedure follows the barcode correction procedure in Cell Ranger by 10X Genomics. Some modifications were made to support ambiguous bases and prevent floating-point underflow. https://kb.10xgenomics.com/hc/en-us/articles/115003822406-How-does-Cell-Ranger-correct-barcode-sequencing-errors

---

Note: Only hamming distance is supported (not Levenshtein distance).

Parameters

- **sequences** – List of sequence strings
- **qualities** – List of quality strings or list of array of qualities (as returned by `pysam.qualitystring_to_array()`)
- **whitelist** – List of whitelist sequences to correct to
- **d** – Hamming distance threshold. Sequences will be corrected to the whitelist with hamming distance <= d. Defaults to 1.
- **confidence** – When a sequence has the same minimum hamming distance to multiple whitelisted sequences, the sequence is assigned to the best whitelisted sequence (using prior probabilities) if the likelihood ratio of this and the sum of all likelihoods is greater than or equal to this value. Defaults to 0.95.
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to True.

Raises
SequenceError – If all the lengths of each sequence, qualitiy and whitelisted sequence are not

equal, the number of sequences and qualities provided are not equal or the whitelist contains duplicates.

Returns
The corrections as a list of whitelisted sequences. For sequences that could not be corrected, the corresponding position contains None.

ngs_tools.sequence.**correct_sequences_to_whitelist_simple**(*sequences: List[str]*, *whitelist: List[str]*, *d: int = 1*, *n_threads: int = 1*, *show_progress: bool = False*) → Dict[str, Union[str, None]]

Correct a list of sequences to a whitelist within *d* hamming distance. Note that *sequences* can contain duplicates, but *whitelist* can not. Unlike *correct_sequences_to_whitelist()*, this function takes a naive approach and discards any sequences that can be corrected to multiple whitelisted sequences.

Parameters

- **sequences** – List of sequence strings
- **whitelist** – List of whitelist sequences to correct to
- **d** – Hamming distance threshold. Sequences will be corrected to the whitelist with hamming distance <= d. Defaults to 1.
- **n_threads** – Number of threads to use. Defaults to 1.
- **show_progress** – Whether to display a progress bar. Defaults to True.

Raises
SequenceError – If all the lengths of each sequence, qualitiy and whitelisted sequence are not equal, the number of sequences and qualities provided are not equal or the whitelist contains duplicates.

Returns
The corrections as a dict of sequence to correction mappings. Note that the return type is different from *correct_sequences_to_whitelist()*.

1.2.5 ngs_tools.utils

Module Contents

Classes

suppress_stdout_stderr	A context manager for doing a "deep suppression" of stdout and stderr in
ParallelWithProgress	Wrapper around joblib.Parallel that uses tqdm to print execution progress.
TqdmUpTo	Wrapper around `tqdm()` so that it can be used with `urlretrieve()`.
FileWrapper	Generic wrapper class for file-formats. Used to wrap file-format-specific

Functions

retry(→ Any)	Utility function to retry a function some number of times, with optional
retry_decorator(→ Callable)	Function decorator to retry a function on exceptions.
run_executable(→ Union[subprocess.Popen, ...)	Execute a single shell command.
is_remote(→ bool)	Check if a string is a remote URL.
is_gzip(→ bool)	Check if a file is Gzipped by checking the magic string.
open_as_text(→ TextIO)	Open a (possibly gzipped) file in text mode.
decompress_gzip(→ str)	Decompress a gzip file to provided file path.
compress_gzip(→ str)	Compress a file into gzip.
concatenate_files(*paths, out_path)	Concatenates an arbitrary number of files into one file.
concatenate_files_as_text(→ str)	Concatenates an arbitrary number of files into one TEXT file.
download_file(→ str)	Download a remote file to the provided path while displaying a progress bar.
stream_file(→ str)	A context manager that creates a FIFO file to use for piping remote files
all_exists(→ bool)	Check whether all provided paths exist.
mkstemp([dir, delete])	Wrapper for `tempfile.mkstemp()` that automatically closes the OS-level
write_pickle(→ str)	Pickle a Python object and compress with Gzip.
read_pickle(→ object)	Load a Python pickle that was compressed with Gzip.
flatten_dictionary(→ Generator[Tuple[tuple, object], ...)	Generator that flattens the given dictionary into 2-element tuples
flatten_iter(→ Generator[object, None, None])	Generator that flattens the given iterable, except for strings.
merge_dictionaries(→ dict)	Merge two dictionaries, applying an arbitrary function *f* to duplicate keys.
flatten_dict_values(→ list)	Extract all values from a nested dictionary.
set_executable(path)	Set the permissions of a file to be executable.

class ngs_tools.utils.**suppress_stdout_stderr**

A context manager for doing a "deep suppression" of stdout and stderr in Python, i.e. will suppress all print, even if the print originates in a compiled C/Fortran sub-function.

This will not suppress raised exceptions, since exceptions are printed

to stderr just before a script exits, and after the context manager has exited (at least, I think that is why it lets exceptions through). https://github.com/facebook/prophet/issues/223

__enter__()

__exit__(*_)

ngs_tools.utils.**retry**(*function: Callable*, *retries: int*, *args: Optional[tuple] = None*, *kwargs: Optional[dict] = None*, *retry_every: Optional[int] = None*, *backoff: bool = False*, *exceptions: Optional[Tuple[Exception]] = None*) → Any

Utility function to retry a function some number of times, with optional exponential backoff.

Parameters

- **function** – Function to retry
- **retries** – Number of times to retry

- **args** – Function arguments
- **kwargs** – Dictionary of keyword arguments
- **retry_every** – Time to wait in seconds between retries. Defaults to no wait time.
- **backoff** – Whether or not to exponential backoff between retries
- **exceptions** – Tuple of exceptions to expect. Defaults to all exceptions.

Returns
Whatever `function` returns

ngs_tools.utils.**retry_decorator**(*retries: int*, *retry_every: Optional[int] = None*, *backoff: bool = False*, *exceptions: Optional[Tuple[Exception]] = None*) → Callable

Function decorator to retry a function on exceptions.

Parameters

- **retries** – Number of times to retry
- **retry_every** – Time to wait in seconds between retries. Defaults to no wait time.
- **backoff** – Whether or not to exponential backoff between retries
- **exceptions** – Tuple of exceptions to expect. Defaults to all exceptions.

ngs_tools.utils.**run_executable**(*command: List[str]*, *stdin=None*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, *wait: bool = True*, *stream: bool = True*, *quiet: bool = False*, *returncode: int = 0*, *alias: bool = True*) → Union[subprocess.Popen, Tuple[subprocess.Popen, List[str], List[str]]]

Execute a single shell command.

Parameters

- **command** – A list representing a single shell command
- **stdin** – Object to pass into the `stdin` argument for :class:`subprocess.Popen`. Defaults to None
- **stdout** – Object to pass into the *stdout* argument for :class:`subprocess.Popen` Defaults to `subprocess.PIPE`
- **stderr** – Object to pass into the *stderr* argument for :class:`subprocess.Popen`, Defaults to `subprocess.PIPE`
- **wait** – Whether to wait until the command has finished. Defaults to True
- **stream** – Whether to stream the output to the command line. Defaults to True
- **quiet** – Whether to not display anything to the command line and not check the return code. Defaults to False
- **returncode** – The return code expected if the command runs as intended. Defaults to 0
- **alias** – Whether to use the basename of the first element of *command*. Defaults to *True*

Returns
A tuple of (the spawned process, string printed to stdout, string printed to stderr) if *wait=True*. Otherwise, just the spawned process.

Raises
subprocess.CalledProcessError – If not `quiet` and the process exited with an exit code != `exitcode`

class ngs_tools.utils.**ParallelWithProgress**(*pbar: Optional[tqdm.tqdm] = None*, *total: Optional[int] = None*, *desc: Optional[str] = None*, *disable: bool = False*, **args*, ***kwargs*)

Bases: joblib.Parallel

Wrapper around joblib.Parallel that uses tqdm to print execution progress. Taken from https://stackoverflow.com/a/61900501

__call__(**args*, ***kwargs*)

print_progress()

ngs_tools.utils.**is_remote**(*path: str*) → bool

Check if a string is a remote URL.

Parameters
path – string to check

Returns
True or False

ngs_tools.utils.**is_gzip**(*path: str*) → bool

Check if a file is Gzipped by checking the magic string.

Parameters
path – path to file

Returns
True or False

ngs_tools.utils.**open_as_text**(*path: str*, *mode: typing_extensions.Literal[r, w]*) → TextIO

Open a (possibly gzipped) file in text mode.

Parameters

- **path** – Path to file
- **mode** – Mode to open file in. Either `r` for read or `w` for write.

Returns
Opened file pointer that supports `read` and `write` functions.

ngs_tools.utils.**decompress_gzip**(*gzip_path: str*, *out_path: str*) → str

Decompress a gzip file to provided file path.

Parameters

- **gzip_path** – Path to gzip file
- **out_path** – Path to decompressed file

Returns
Path to decompressed file

ngs_tools.utils.**compress_gzip**(*file_path: str*, *out_path: str*) → str

Compress a file into gzip.

Parameters

- **file_path** – Path to file
- **out_dir** – Path to compressed file

Returns
Path to compressed file

ngs_tools.utils.**concatenate_files**(**paths: str*, *out_path: str*)

Concatenates an arbitrary number of files into one file.

Parameters

- ***paths** – An arbitrary number of paths to files
- **out_path** – Path to place concatenated file

Returns
Path to concatenated file

ngs_tools.utils.**concatenate_files_as_text**(**paths: str*, *out_path: str*) → str

Concatenates an arbitrary number of files into one TEXT file.

Only supports plaintext and gzip files.

Parameters

- ***paths** – An arbitrary number of paths to files
- **out_path** – Path to place concatenated file

Returns
Path to concatenated file

class ngs_tools.utils.**TqdmUpTo**

Bases: tqdm.tqdm

Wrapper around tqdm() so that it can be used with urlretrieve(). https://github.com/tqdm/tqdm/blob/master/examples/tqdm_wget.py

update_to(*b=1*, *bsize=1*, *tsize=None*)

ngs_tools.utils.**download_file**(*url: str*, *path: str*) → str

Download a remote file to the provided path while displaying a progress bar.

Parameters

- **url** – Remote url
- **path** – Local path to download the file to

Returns
Path to downloaded file

ngs_tools.utils.**stream_file**(*url: str*, *path: str*) → str

A context manager that creates a FIFO file to use for piping remote files into processes. This function must be used as a context manager (the with keyword) so that any exceptions in the streaming thread may be captured.

This function spawns a new thread to download the remote file into a FIFO file object. FIFO file objects are only supported on unix systems.

Parameters

- **url** – Url to the file
- **path** – Path to place FIFO file

Yields
Path to FIFO file

Raises
OSError – If the operating system does not support FIFO

ngs_tools.utils.**all_exists**(**paths: str*) → bool

Check whether all provided paths exist.

Parameters
***paths** – paths to files

Returns
True if all files exist, False otherwise

class ngs_tools.utils.**FileWrapper**(*path: str*, *mode: typing_extensions.Literal[r, w] = 'r'*)

Generic wrapper class for file-formats. Used to wrap file-format-specific implementations of reading and writing entries. This class is not designed to be initialized directly. Instead, it should be inherited by children that implements the `read` and `write` methods appropriately.

The file is opened immediately as soon as the class is initialized. This class can also be used as a context manager to safely close the file pointer with a `with` block.

path
Path to the file

mode
Open mode. Either `r` or `w`.

fp
File pointer

closed
Whether the file has been closed

property is_remote: bool

property is_gzip: bool

property closed: bool

__del__()

__enter__()

__exit__(**args*, ***kwargs*)

__iter__()

_open()
Open the file

close()
Close the (possibly already-closed) file

reset()
Reset this wrapper by first closing the file and re-running initialization, which re-opens the file.

tell() → int
Get the current location of the file pointer

abstract read() → Any
Read a single entry. This method must be overridden by children.

abstract write(*entry: Any*)

Write a single entry. This method must be overridden by children.

ngs_tools.utils.**mkstemp**(*dir: Optional[str] = None*, *delete: bool = False*)

Wrapper for `tempfile.mkstemp()` that automatically closes the OS-level file descriptor. This function behaves like `tempfile.mkdtemp()` but for files.

Parameters

- **dir** – Directory to create the temporary file. This value is passed as the `dir` kwarg of `tempfile.mkstemp()`. Defaults to None.
- **delete** – Whether to delete the temporary file before returning. Defaults to False.

Returns

path to the temporary file

ngs_tools.utils.**write_pickle**(*obj: object*, *path: str*, **args*, ***kwargs*) → str

Pickle a Python object and compress with Gzip.

Any additional arguments and keyword arguments are passed to `pickle.dump()`.

Parameters

- **obj** – Object to pickle
- **path** – Path to save pickle

Returns

Saved pickle path

ngs_tools.utils.**read_pickle**(*path: str*) → object

Load a Python pickle that was compressed with Gzip.

Parameters

path – Path to pickle

Returns

Unpickled object

ngs_tools.utils.**flatten_dictionary**(*d: dict*, *keys: Optional[tuple] = None*) → Generator[Tuple[tuple, object], None, None]

Generator that flattens the given dictionary into 2-element tuples containing keys and values. For nested dictionaries, the keys are appended into a tuple.

Parameters

- **d** – Dictionary to flatten
- **keys** – Previous keys, defaults to None. Used exclusively for recursion.

Yields

Flattened dictionary as (keys, value)

ngs_tools.utils.**flatten_iter**(*it: Iterable*) → Generator[object, None, None]

Generator that flattens the given iterable, except for strings.

Parameters

lst – Iterable to flatten

Yields

Flattened iterable elements

ngs_tools.utils.**merge_dictionaries**(*d1: dict*, *d2: dict*, *f: Callable[[object, object], object] = add*, *default: object = 0*) → dict

Merge two dictionaries, applying an arbitrary function *f* to duplicate keys. Dictionaries may be nested.

Parameters

- **d1** – First dictionary
- **d2** – Second dictionary
- **f** – Merge function. This function should take two arguments and return one, defaults to +
- **default** – Default value or callable to use for keys not present in either dictionary, defaults to *0*

Returns

Merged dictionary

ngs_tools.utils.**flatten_dict_values**(*d: dict*) → list

Extract all values from a nested dictionary.

Parameters

d – Nested dictionary from which to extract values from

Returns

All values from the dictionary as a list

ngs_tools.utils.**set_executable**(*path: str*)

Set the permissions of a file to be executable.

1.3 Package Contents

ngs_tools.**__version__ = '1.8.4'**

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W