Pathway Analysis Documentation

Release 0.1

KNBiBS

Nov 11, 2018

Contents:

CHAPTER 1

Command Line Interface

1.1 User guide

The command line interface has built in help. To display the help, please append *-h* to the program call, for example:

```
./patapy.py -h
```

The help option responds to arguments your provide, so you can get details about your method of choice with:

```
./patapy.py gsea -h
```

where gsea is the name of a method; likewise, you can display help for any of samples specification options (case/control/data), e.g.:

```
./patapy.py control -h
```

1.2 Predefined parsers

Parsers are defined in *command_line.main* module.

class CLI (*parser_name=None*, ***kwargs*)

The main parser, the one exposed directly to the user.

parse_args (*args*)

Same as `parse_known_args()` but all arguments must be parsed.

This is an equivalent of `argparse.ArgumentParser.parse_args()` although it does >not< support *namespace* keyword argument.

Comparing to `parse_known_args()`, this method handles help messages nicely (i.e. passes everything to `argparse`).

Parameters **args** – strings to parse, default is sys.argv[1:]

produce(*unknown_args*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

class CLIExperiment(*parser_name=None*, ***kwargs*)
Use both: case and control or data to create an Experiment.

produce(*unknown_args=None*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

class PhenotypeFactory(*parser_name=None*, ***kwargs*)
Provide {parser_name} samples. Requires a file (or files) with samples.

The files should come in Delimiter Separated Values format (like .csv or .tsv). The default delimiter is a tab character. The first column of each file should contain gene identifiers.

To use only a subset of samples from files(s) specify column numbers (–columns) or sample names (–samples) of desired samples.

produce(*unknown_args=None*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

class SingleFileExperimentFactory(*parser_name=None*, ***kwargs*)
Provide both: case and control samples from a single file.

This is just a shortcut for specifying the same file for both: case and control samples sets. You have to provide –case or –control (or both) to specify which columns contain controls.

If you specify only one of –case and –control, it will be assumed that all other columns should be used for the other set of samples (if you use *–case 0,1,2* and your file has five columns with samples, then columns three and four will be used to create control samples).

To enable more advanced features, please use *control'&'case* options (instead of the currently selected *data* sub-parser).

produce(*unknown_args=None*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

1.3 Creating custom arguments and parsers

Please use *command_line.parser* module to create custom parsers and arguments.

class Argument(*name=None*, *short=None*, *optional=True*, *as_many_as=None*, ***kwargs*)

Defines argument for *Parser*.

In essence, this is a wrapper for `argparse.ArgumentParser.add_argument()`, so most options (type, help) which work in standard Python parser will work with Argument too. Additionally, some nice features, like automated naming are available.

Worth to mention that when used with `MethodParser`, *type* and *help* will be automatically deduced.

class Parser(*parser_name=None*, ***kwargs*)

Parser is a wrapper around Python built-in `argparse.ArgumentParser`.

Subclass the *Parser* to create your own parser.

Use help, description and epilog properties to adjust the help screen. By default help and description will be auto-generated using docstring and defined arguments.

Attach custom arguments and sub-parsers by defining class-variables with *Argument* and *Parser* instances.

Example:

```
class TheParser(Parser):
    help = 'This takes only one argument, but it is required'

    arg = Argument(optional=False, help='This is required')

class MyParser(Parser):
    description = 'This should be a longer text'

    my_argument = Argument(type=int, help='some number')
    my_sub_parser = TheParser()

    epilog = 'You can create a footer with this'

# To execute the parser use:

parser = MyParser()

# The commands will usually be `sys.argv[1:]`
commands = '--my_argument 4 my_sub_parser value'.split()

namespace = parser.parse_args(commands)

# `namespace` is a normal `argparse.Namespace`
assert namespace.my_argument == 4
assert namespace.my_sub_parser.arg == 'value'
```

Implementation details:

To enable behaviour not possible with limited, plain *ArgumentParser* (e.g. to dynamically attach a sub-parser, or to chain two or more sub-parsers together) the stored actions and sub-parsers are:

- not attached permanently to the parser,
- attached in a tricky way to enable desired behaviour,
- executed directly or in hierarchical order.

Class-variables with parsers will be deep-copied on initialization, so you do not have to worry about re-use of parsers.

attach_argument(*argument*, *parser=None*)
Attach Argument instance to given (or own) argparse.parser.

attach_subparsers()
Only in order to show a nice help, really.

There are some issues when using subparsers added with the built-in add_subparsers for parsing. Instead subparsers are handled in a custom implementation of parse_known_args (which really builds upon the built-in one, just tweaking some places).

bind_argument(*argument*, *name=None*)
Bind argument to current instance of Parser.

bind_parser(*parser*, *name*)
Bind deep-copy of Parser with this instance (as a sub-parser).

Parameters

- **parser** (*Parser*) – parser to be bound as a sub-parser (must be already initialized)
- **name** – name of the new sub-parser

This method takes care of 'translucent' sub-parsers (i.e. parsers which expose their arguments and sub-parsers to namespace above), saving their members to appropriate dicts (lifted_args/parsers).

description
Longer description of the parser.

Description is shown when user narrows down the help to the parser with: `./run.py sub_parser_name -h`.

epilog
Use this to append text after the help message

error(*message*)
Raises SystemExit with status code 2 and shows usage message.

help
A short message, shown as summary on >parent< parser help screen.

Help will be displayed for sub-parsers only.

parse_args(*args=None*)
Same as *parse_known_args()* but all arguments must be parsed.

This is an equivalent of `argparse.ArgumentParser.parse_args()` although it does >not< support *namespace* keyword argument.

Comparing to *parse_known_args()*, this method handles help messages nicely (i.e. passes everything to `argparse`).

Parameters **args** (`Optional[Sequence[str]]`) – strings to parse, default is sys.argv[1:]

parse_known_args(*args*)
Parse known arguments, like `argparse.ArgumentParser.parse_known_args()`.

Additional features (when compared to argparse implementation) are:

- ability to handle multiple sub-parsers
- validation with *self.validate* (run after parsing)
- additional post-processing with *self.produce* (after validation)

produce(*unknown_args*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

pull_to_namespace_above
Makes the parser "translucent" for the end user.

Though parsing methods (as well as validate & produce) are still evaluated, the user won't be able to see this sub-parser in command-line interface.

This is intended to provide additional logic separation layer & to keep the parsers nicely organized and nested, without forcing the end user to type in prolonged names to localise an argument in a sub-parser of a sub-parser of some other parser.

validate(*opts*)
Perform additional validation, using *Argument.validate*.

As validation is performed after parsing, all arguments should be already accessible in *self.namespace*. This enables testing if arguments depending one on another have proper values.

dedent_help(*text*)
Dedent text by four spaces

group_arguments(*args*, *group_names*)
Group arguments into given groups + None group for all others

CHAPTER 2

Methods

2.1 Implemented methods

class GSEA
Not finished yet.

2.2 Adding a new Method

To implement a new method and integrate it with the Command Line Interface provided by this package, please inherit from *Method* class.

class Method
Defines method of pathway analysis & its arguments.

Simple arguments (like `threshold`) can be simply defined as arguments and keyword arguments of *__init__*.

For example:

```
class MyMethod(Method)
    def __init__(threshold:float=0.05):
        pass
```

For the simple arguments following information will be deduced:

- type: will be retrieved from type annotations; currently only non-abstract types (int, str, float and so on) are supported. We can implement abstract types from *typing* if needed.
- default: from keyword arguments.
- help: will be retrieved from docstrings

If you need more advanced options (like aggregation), or just do not like having a mess in your *__init__* signature, please define the arguments in body of your class using *Argument* constructor.

For example:

```
class MyMethod(Method):

    database = Argument(
        type=argparse.FileType('r'),
        help='Path to file with the database'
    )

    def __init__(threshold:float=0.05, database=None):
        pass
```

If help is given in both *Argument* and docstring, then the help from *Argument()* takes precedence over the help in docstrings (as docstrings should cover not only CLI usage but also describe how to use the method as a standalone object - to enable advanced users to customize methods).

help
: Return string providing help for this method.

  The help message shows up when *./run method_name -h*.

name
: Return method name used internally and in command line interface.

  The name should not include any spaces.

CHAPTER 3

Biological Objects

class Gene(*name*, *description=None*)

Stores gene's identifier and description (multiton).

At a time there can be only one gene with given identifier, i.e. after the first initialization, all subsequent attempts to initialize a gene with the same identifier will return exactly the same object. This is so called multiton pattern.

Example

```
>>> x = Gene('TP53')
>>> y = Gene('TP53')
>>> assert x is y   # passes, there is only one gene
```

class Phenotype(*name*, *samples=None*)

Phenotype is a collection of samples of common origin or characteristic.

An example phenotype can be: (Breast_cancer_sample_1, Breast_cancer_sample_2) named "Breast cancer".

The common origin/characteristics for "Breast cancer" phenotype could be "a breast tumour", though samples had been collected from two donors.

Another example are controls: (Control_sample_1, Control_sample_2) named "Control".

The common characteristic for these samples is that both are controls.

as_array()

Returns: `pandas.DataFrame` object with data for all samples.

classmethod from_file(*name*, *file_object*, *columns_selector=None*, *samples=None*, *delimiter='t'*, *index_col=0*, *use_header=True*, *reverse_selection=False*, *prefix=None*, *header_line=0*, *description_column=None*)

Create a phenotype (collection of samples) from csv/tsv file.

Parameters

- **name** – a name of the phenotype (or group of samples) which will identify it (like "Tumour_1" or "Control_in_20_degrees")

- **file_object** – a file (containing gene expression) of the following structure:
  - names of samples separated by a tab in the first row,
  - gene symbol/name followed by gene expression values for every sample in remaining rows;

  an additional column "description" is allowed between genes column and sample columns, though it has to be explicitly declared with *description_column* argument.
- **columns_selector** (Optional[Callable[[Sequence[int]], Sequence[int]]]) – a function which will select (and return) a subset of provided column identifiers (do not use with *samples*)
- **samples** – a list of names of samples to extract from the file (do not use with *columns_selector*)
- **reverse_selection** – if you want to use all columns but the selected ones (or all samples but the selected) set this to True
- **delimiter** (str) – the delimiter of the columns
- **index_col** (int) – column to use as the gene names
- **use_header** – does the file have a header?
- **prefix** – prefix for custom samples naming schema
- **header_line** – number of non-empty line with sample names
- **description_column** – is column with description of present in the file (on the second position, after gene identifiers)?

classmethod from_gsea_file()
Stub: if we need to handle very specific files, for various analysis methods, we can extend Phenotype with class methods like from_gsea_file.

class Sample(*name*, *data*)
Sample contains expression values for genes.

as_array()
Returns: one-dimensional labeled array with Gene objects as labels

classmethod from_array(*name*, *panda_series*, *descriptions=False*)
Create a sample from pd.Series or equivalent.

Parameters

- **name** – name of the sample
- **panda_series** (Series) – series object where columns represent values of genes and names are either gene identifiers of tuples: `(gene_identifier, description)`
- **descriptions** – are descriptions present in names of the series object?

classmethod from_names(*name*, *data*)
Create a sample from a gene_name: value mapping.

Parameters

- **name** – name of sample
- **data** (Mapping[str, float]) – mapping (e.g. dict) where keys represent gene names

CHAPTER 4

Statistics Utilities

ttest_ind_phenotype(*case*, *control*, *alternative='two-sided'*)

Two sided t-test of case sample(s) and mean expression values in base samples across all genes :type case: Union[*Phenotype*, *Sample*] :param case: either Sample of Phenotype object with case sample(s) :type control: Union[*Phenotype*, *Sample*] :param control: either Sample of Phenotype object with control sample(s) :param alternative: string with the alternative hypothesis, H1, has to be one of the following:

> 'two-sided': H1: difference in means not equal to value (default) 'larger' : H1: difference in means larger than value 'smaller' : H1: difference in means smaller than value

Returns: tstat [float or numpy array in case of multiple case samples - test statisic] pvalue : float or numpy array in case of multiple case samples - pvalue of the t-test df : int or float - degrees of freedom used in the t-test

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

m

s

Index