
Pynion Documentation

Release 0.0.1

Jaume Bonet

January 29, 2016

Pynion is a small python minion library that provides two key features to help both developers and python users alike:

- For **developers**, it provides a series of classes designed to help logging as well as in the control of I/O processes.
- For **users**, it provides a system to build a project's pipeline by tracking multiple python executions.

Contents

1.1 Install

The easiest way to install **Pynion** is through pip:

```
pip install pynion
```

Alternatively, the source code can be directly downloaded at the [github repository](#).

Once decompressed, the library can be installed with:

```
python setup.py install
```

1.2 Quickstart

1.2.1 On the main script

1.2.2 On a user defined library

1.3 API documentation

1.3.1 Classes

Manager

class `pynion.Manager`

The Manager class is a *Singleton* that crosses through the entire library. It is the main controller of the user's preferences.

`__init__()`

The class is instantiated at some point during the load of the library's code. Thus, no parameters are passed to it. As any new instantiation will only call the initial instance, parameters cannot be passed afterward.

Although any attribute can be accessed through its given *setter*, default values can be assigned to the first initialization of **Manager** through a configuration file. The configuration file should have the following parameters:

```
[manager]
stdout    = False
verbose   = False
debug     = False
detail    = False
overwrite = False
unclean   = False
logfile   =

[project]
name      = _info-project.json
config    = _config-project.json
pipeline  = _pipeline-project.json
```

The user's configuration file must then be linked to a system variable named `PYNION_CONFIG_PY`. Thus, the configuration file can be setted globally for all executions directly from bash,

```
export PYNION_CONFIG_PY=user.settings
```

Or specifically to a script by

```
import os
os.environ["PYNION_CONFIG_PY"] = 'user.settings'
```

BEFORE importing the **Pynion** library.

Regarding the different execution parameters in the configuration file:

Parameters

- **stdout** (*bool*) – Create a `logging.StreamHandler` for standard output.
- **verbose** (*bool*) – Activate level *verbose* of logging report.
- **debug** (*bool*) – Activate level *debug* of logging report. It also forces the activation of *verbose*.
- **detail** (*bool*) – Activate level *detail* of logging report. It also forces the activation of *debug*, *verbose* and *unclean*.
- **overwrite** (*bool*) – When `True`, allows overwriting existing files.
- **unclean** (*bool*) – When `True`, avoids the deletion of temporary and empty files at the end of the execution.
- **logfile** (*bool*) – When defined, it creates a `logging.StreamHandler` to a file with the provided name. If `logfile = default` or a directory, it creates a logfile with a predefined name that includes the name of the execution and the pid of the process.

Regarding the project parameters in the configuration file: **(TODO)**

set_verbose()

Activate level *verbose* of logging report

set_debug()

Activate level *debug* of logging report. It also forces the activation of *verbose*.

set_detail()

Activate level *detail* of logging report. It also forces the activation of *debug*, *verbose* and *unclean*.

set_unclean()

Avoids the deletion of temporary and empty files at the end of the execution.

set_stdout()

Create a `logging.StreamHandler` for standard output.

set_overwrite()

Allows overwriting existing files.

set_logfile (*logname*='~/home/docs/checkouts/readthedocs.org/user_builds/pynion/checkouts/stable/docs/source')

Creates a `logging.StreamHandler` to a file.

Parameters *logname* (*str*) – Name of the output logging file. If *logname* is a directory, it will create a logging file in that directory named by the name of the execution and its pid. Default value is current working directory.

is_verbose()

Assess the status of the *verbose* logging level

is_debug()

Assess the status of the *debug* logging level

is_detail()

Assess the status of the *detail* logging level

add_temporary_file (*tempfile*)

Register a new temporary file.

Parameters *tempfile* (*str*) – Name of the temporary file.

add_experiment_file (*filename*, *action*)

Register a new experiment file.

Parameters

- **filename** (*str*) – Name of the experiment file.
- **action** (*str*) – Open mode of the registered file ('r', 'w', 'a')

add_citation (*citation*)

Adds a new citation from some code or other to be printed at the end of the execution.

Parameters *citation* (*str*) – Citation to store

countdown (*max_time*)

Generate a STDERR printed countdown when needed to wait for something.

Parameters *max_time* (*int*) – Time to wait, in seconds.

evaluate_overwrite (*overwrite*=None)

Given a overwrite command, it evaluates it with the global overwrite configuration.

Parameters *overwrite* (*bool*) – Particular overwrite status. Default is `None`

Returns Final overwrite status

Return type `bool`

info (*mssg*)

Print *verbose* level information.

Parameters *mssg* (*str*) – Message to relay through logging. If it is `list`, each position is treated as a new line.

debug (*mssg*)

Print *debug* level information.

Parameters `mssg` (*str*) – Message to relay through logging. If it is `list`, each position is treated as a new line.

detail (*mssg*)

Print *detail* level information.

Parameters `mssg` (*str*) – Message to relay through logging. If it is `list`, each position is treated as a new line.

warning (*mssg*)

Print *warning*.

Parameters `mssg` (*str*) – Message to relay through logging. If it is `list`, each position is treated as a new line.

exception (*mssg*)

Print *exceptions* and quit.

Parameters `mssg` (*str*) – Message to relay through logging. If it is `list`, each position is treated as a new line.

File

class `pynion.File`

The **File** object is a *factory object pattern* that creates either a `pynion.filesystem._filetypes.BaseFile`, a `pynion.filesystem._filetypes.CompressedFile` or a `pynion.filesystem._filetypes.ContainerFile`.

Parameters

- **file_name** (*str*) – Name of the file.
- **action** (*str*) – Action to perform to the file ('r', 'w', 'a', 't').
- **overwrite** (*bool*) – Specific overwrite policy over the file. By default, it uses the value of `pynion.Manager` `overwrite`.
- **temp** (*bool*) – Register the file as a temporary file, which means that it will be erased at the end of the execution unless `pynion.Manager` `clean` is set to `False`.
- **pattern** (*str*) – A pattern to match sections of the file name to attributes in the generated object.

Raise `pynion.errors.ffe.FileAccessError` if asked for a file without the right user permissions.

Raise `pynion.errors.ffe.FileDirNotExistError` if asked to write a file in a directory that does not exist.

Raise `pynion.errors.ffe.FileIsDirError` if `path` is a directory.

Raise `pynion.errors.ffe.FileNotExistsError` if asked to read a files that does not exists.

Raise `pynion.errors.ffe.FileOverwriteError` if asked to write a file that already exist and own `overwrite` or `pynion.Manager` `overwrite` are `False`.

Raise `pynion.errors.ffe.FileWrongActionError` when asked for an unknown action.

Raise `pynion.errors.ffe.FileWrongPatternIDError` if names in the pattern match names of attributes that exist in the returned class.

Raise `pynion.errors.ffe.FileWrongPatternFormatError` if the given pattern cannot be matched to the file name.

BaseFile

class `pynion.filesystem._filetypes.BaseFile` (*file_name, action*)

The **BaseFile** `pynion.Multiton` is a file management object created directly through the `py:class:pynion.File` factory.

It specifically manages regular files.

full

Returns Full path of the file

Return type `str`

dir

Returns Full path containing directory

Return type `str`

last_dir

Returns Name of the containing directory

Return type `str`

name

Returns Name of the file

Return type `str`

prefix

Returns Name of the file without extension

Return type `str`

first_prefix

Returns Name of the first section of the file

Return type `str`

extension

Returns Name of the file's extension

Return type `str`

extensions

Returns List of all the sections of the file name except the first one.

Return type `list`

descriptor

Returns Descriptor of the stored file

Return type `str`

size

Returns File size

Return type `str`

pattern

Returns Dictionary with the pattern assigned sections of the file name.

Return type `dict`

is_open

Returns Check if the file descriptor is open

Return type `bool`

is_to_write

Returns Check if the file is set to write

Return type `bool`

is_to_read

Returns Check if the file is set to read

Return type `bool`

relative_to (*path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/pynion/checkouts/stable/docs/source')*)

Parameters **path** (*str*) – Path to which the relative path is required.

Returns Actual path relative to the query path

Return type `str`

open()

Open the file in the previously defined action type. `:rtype: self`

read()

Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in write mode.

Return type File Descriptor

readline()

Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in write mode.

Returns One line of the file.

Return type `str`

readJSON (*encoding='utf-8'*)

Retrieve all data in file as a JSON dictionary.

Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in write mode.

Return type `dict`

write (*line*)

Write to the file

Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in read mode.

flush()

Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in read mode.

close()
Close the file.

CompressedFile

class `pynion.filesystem._filetypes.CompressedFile` (*file_name, action, ctype*)
The **CompressedFile** `pynion.Multiton` is a file management object created directly through the `py:class:pynion.File` factory.

Extends `pynion.filesystem._filetypes.BaseFile`

It specifically manages compressed files.

is_gzipped
Returns Check if compression is gzip
Return type `bool`

is_bzipped
Returns Check if compression is bzip
Return type `bool`

open()
Open the file in the previously defined action type. `:rtype: self`

flush()
Raise `pynion.errors.fe.FileWrongRequestedActionError` if opened in read mode.

ContainerFile

class `pynion.filesystem._filetypes.ContainerFile` (*file_name, action, ctype*)
The **CompressedFile** `pynion.Multiton` is a file management object created directly through the `py:class:pynion.File` factory.

Extends `pynion.filesystem._filetypes.BaseFile`

It specifically manages compacted or compressed files with multiple files within.

is_gzipped
Returns Check if compression is gzip
Return type `bool`

is_bzipped
Returns Check if compression is bzip
Return type `bool`

is_zipped
Returns Check if compression is zip
Return type `bool`

is_tarfile

Returns Check if compression is tar

Return type `bool`

open()

Open the file in the previously defined action type. :rtype: self

read_file(file_name)

Get a specific file from the file bundle.

Parameters **file_name** (`str`) – Name of the query internal file.

Raise `pynion.errors.fe.FileContainerFileNotFound` if query file is not in the bundle.

Return type `str`

has_file(file_name)

Query if file exists in the bundle.

Return type `bool`

extract(target_file=None, target_dir='/home/docs/checkouts/readthedocs.org/user_builds/pynion/checkouts/stable/docs/source')

Extract a specific file from the bundle.

Parameters

- **target_file** (`str`) – Query file to extract from the bundle. If `None`, all files contained
- **target_dir** (`str`) – Directory to which to write the file. By default that is the current working directory

Raise `pynion.errors.fe.FileContainerFailedExtraction` if target_file not

list_files()

Name of all files in the bundle.. :rtype: list

length()

Number of lines in the file. :rtype: int

Path

class `pynion.Path(name)`

The `Path` `pynion.Multiton` is an extension (not an actual inheritance) from the `pathlib.Path`.

__init__(name)

Initializes the Path object. If the directory does not exist, it is created.

Parameters **name** (`str`) – name of the directory.

Raise `pynion.errors.pe.PathIsFile` if the given name is a file.

full

Returns Full path of the directory

Return type `str`

parent

Returns Name of parent directory

Return type `str`

parents

Returns Name of all parent directories

Return type `list`

name

Returns Name of the directory

Return type `str`

relative_to (*path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/pynion/checkouts/stable/docs/source')*)

Parameters **path** (`str`) – Path to which the relative path is required.

Returns Actual path relative to the query path

Return type `str`

makedirs()

Create the directory of the path.

Command is ignored if the directory already exists. Required parent directories are also created.

list_directories (*rootless=False*)

List all the directories inside the requested path.

Parameters **rootless** (`bool`) – When `True`, the directories are returned relative to the path. Full path otherwise.

Return type `iterator`

list_files (*pattern='*', avoid_empty_files=True, rootless=False*)

List all the files inside the requested path.

Parameters

- **pattern** (`str`) – Pattern to match the files (bash `ls` format).
- **avoid_empty_files** (`bool`) – When `True`, empty files are omitted.
- **rootless** (`bool`) – When `True`, the files are returned relative to the path. Full path otherwise.

Return type `iterator`

do_files_match (*pattern='*', avoid_empty_files=True*)

Search if files inside the directory match a given pattern.

Parameters

- **pattern** (`str`) – Pattern to match the files (bash `ls` format).
- **avoid_empty_files** (`bool`) – When `True`, empty files are omitted.

Return type `bool`

compare_to (*other, by_dir=True, by_file=False, as_string=False*)

Compare the directory to another path.

Parameters

- **other** (`str`) – Path to compare to.
- **by_dir** (`bool`) – Comparison by directories.
- **by_file** (`bool`) – Comparison by file. If `True`, overrides `by_dir`.

- **as_string** (*bool*) – Formats the returned dict as a string.

Return type *dict*

sync_to (*other*, *by_dir=True*, *by_file=False*)

Sync from directory to another path.

Parameters

- **other** (*str*) – Path to sync with.
- **by_dir** (*bool*) – Sync by directories.
- **by_file** (*bool*) – Sync by file. If *True*, overrides *by_dir*.

sync_from (*other*, *by_dir=True*, *by_file=False*)

Sync from another path to the directory.

Parameters

- **other** (*str*) – Path to sync with.
- **by_dir** (*bool*) – Sync by directories.
- **by_file** (*bool*) – Sync by file. If *True*, overrides *by_dir*.

Executable

class `pynion.Executable` (*executable*, *path=None*)

Manages the execution of external programs.

Parameters

- **executable** (*str*) – name of the executable program
- **path** (*str*) – path to the executable. Not needed if the executable is in the `$PATH` environment variable. Default is *None*

Raise `pynion.errors.xe.ExecutableNoExistsError` if the executable does not exist

Raise `pynion.errors.xe.ExecutableNotInPathError` if the path is *None* and the executable is not in the `$PATH` environment variable.

Raise `pynion.errors.xe.ExecutablePermissionDeniederror` if executable has no execution permission

executable

Name of the executable.

Return type *str*

path

Path to the executable.

Return type *str*

command

Full command to execute.

Return type *list*

full_executable

Full executable with path.

Return type *str*

add_attribute (*attribute_value*, *attribute_id=None*)

Adds a new parameter to the command. Specifically for parameters with ‘tags’ like ‘-i’.

Parameters

- **attribute_value** (*str*) – value of the attribute to add
- **attribute_id** (*str*) – label of the attribute to add. By default is `None`, which makes the function identical to `pynion.Executable.add_parameter`.

add_parameter (*parameter*)

Adds a new stand alone parameter to the command.

Parameters **parameter** (*str*) – value of the parameter to add

clean_command ()

Removes all attributes and parameters added to the command.

backup_command ()

Store a copy of the command up to that point to retrieve import afterwards.

restore_command ()

Retrieve the backup command into the working command. The backup command is emptied.

execute (*silent=False*)

Executes the commands.

Parameters **silent** (*bool*) – If `True`, external program STDERR is shown through STDERR

Raises `SystemError` if an error occurs in the external program.

1.3.2 Metaclasses

Metaclasses in **pynion** are designed in order to help users develop their classes by adding a certain build behavior to the derived classes.

Singleton

class `pynion.Singleton`

The **singleton pattern** is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

As a metaclass, the pattern is applied to derived classes such as

```
from pynion import Singleton

class Foo(object):
    __metaclass__ = Singleton

    def __init__(self, bar):
        self.bar = bar
```

Derived classes can become parents of other classes. Classes inherited from a `__metaclass__ = Singleton` are also Singleton.

Multiton

class `pynion.Multiton`

The **multiton pattern** is a design pattern similar to the singleton. The multiton pattern expands on the singleton concept to manage a map of named instances as key-value pairs. This means that rather than having a single instance per application the multiton pattern instead ensures a single instance per key.

As a metaclass, the pattern is applied to derived classes through the `__metaclass__`. By default, the key attribute of the multiton is either:

- the `__init__()` named argument **name** or
- the first argument of `__init__()` if not named.

The default named argument can be changed by adding the class attribute `_IDENTIFIER`

```
from pynion import Multiton

class Foo(object):
    __metaclass__ = Multiton
    _IDENTIFIER = 'newID' # by default this will be 'name'

    def __init__(self, newID):
        self.id = newID
```

Derived classes can become parents of other classes. Classes inherited from a `__metaclass__ = Multiton` are also Multiton, although their `_IDENTIFIER` can be changed.

1.3.3 Abstractclasses

Abstract classes in **pynion** are designed in order to help users develop their classes by adding a set of default functions and properties.

JSONer

class `pynion.JSONer`

The abstract class **JSONer** adds to its descendants the required functions in order to export/import any given object as json. This can be useful to share the data between different languages or simply to store pre-calculated data.

to_json (*unpicklable=True, readable=False, api=False*)

Export the object to a json formatted string.

Parameters

- **unpicklable** (*bool*) – When `False` the resulting json cannot be reloaded as the same object again. Makes the json smaller.
- **readable** (*bool*) – When flattening complex object variables to json, this will include a human-readable version together with the stored json. See `jsonpickle` on how to flatten and restore complex variable types. It does not affect the conversion of the json string into the object again.
- **api** (*bool*) – When flattening complex object variables to json, this will substitute the compressed data by the human readable version. Although it might allow for the conversion of the json into the object, some attributes will become their simplest representation. For example, a `numpy array` will be reloaded as a simple python array.

Returns a json representation of the object

Return type `str`

to_dict (*unpicklable=True, readable=False, api=False*)

Export the object to a json as a dictionary.

Parameters

- **unpicklable** (*bool*) – When `False` the resulting json cannot be reloaded as the same object again. Makes the json smaller.
- **readable** (*bool*) – When flattening complex object variables to json, this will include a human-readable version together with the stored json. See [jsonpickle](#) on how to flatten and restore complex variable types. It does not affect the conversion of the json string into the object again.
- **api** (*bool*) – When flattening complex object variables to json, this will substitute the compressed data by the human readable version. Although it might allow for the conversion of the json into the object, some attributes will become their simplest representation. For example, a [numpy array](#) will be reloaded a a simple python array.

Returns a json dictionary object

Return type `dict`

static from_json (*json_data*)

Given a json-formated string, it recreates the object.

Parameters **json_data** (*str*) – json-formated string.

Returns an instance of the caller object type.

Return type `object instance`

p

`pynion`, [1](#)
`pynion.abstractclass`, [14](#)
`pynion.metaclass`, [13](#)

Symbols

`__init__()` (Manager method), 3

`__init__()` (Path method), 10

A

`add_attribute()` (pynion.Executable method), 12

`add_citation()` (pynion.Manager method), 5

`add_experiment_file()` (pynion.Manager method), 5

`add_parameter()` (pynion.Executable method), 13

`add_temporary_file()` (pynion.Manager method), 5

B

`backup_command()` (pynion.Executable method), 13

`BaseFile` (class in `pynion.filesystem._filetypes`), 7

C

`clean_command()` (pynion.Executable method), 13

`close()` (pynion.filesystem._filetypes.BaseFile method), 9

`command` (pynion.Executable attribute), 12

`compare_to()` (pynion.Path method), 11

`CompressedFile` (class in `pynion.filesystem._filetypes`), 9

`ContainerFile` (class in `pynion.filesystem._filetypes`), 9

`countdown()` (pynion.Manager method), 5

D

`debug()` (pynion.Manager method), 5

`descriptor` (pynion.filesystem._filetypes.BaseFile attribute), 7

`detail()` (pynion.Manager method), 6

`dir` (pynion.filesystem._filetypes.BaseFile attribute), 7

`do_files_match()` (pynion.Path method), 11

E

`evaluate_overwrite()` (pynion.Manager method), 5

`exception()` (pynion.Manager method), 6

`Executable` (class in `pynion`), 12

`executable` (pynion.Executable attribute), 12

`execute()` (pynion.Executable method), 13

`extension` (pynion.filesystem._filetypes.BaseFile attribute), 7

`extensions` (pynion.filesystem._filetypes.BaseFile attribute), 7

`extract()` (pynion.filesystem._filetypes.ContainerFile method), 10

F

`File` (class in `pynion`), 6

`first_prefix` (pynion.filesystem._filetypes.BaseFile attribute), 7

`flush()` (pynion.filesystem._filetypes.BaseFile method), 8

`flush()` (pynion.filesystem._filetypes.CompressedFile method), 9

`from_json()` (pynion.JSONer static method), 15

`full` (pynion.filesystem._filetypes.BaseFile attribute), 7

`full` (pynion.Path attribute), 10

`full_executable` (pynion.Executable attribute), 12

H

`has_file()` (pynion.filesystem._filetypes.ContainerFile method), 10

I

`info()` (pynion.Manager method), 5

`is_bzipped` (pynion.filesystem._filetypes.CompressedFile attribute), 9

`is_bzipped` (pynion.filesystem._filetypes.ContainerFile attribute), 9

`is_debug()` (pynion.Manager method), 5

`is_detail()` (pynion.Manager method), 5

`is_gzipped` (pynion.filesystem._filetypes.CompressedFile attribute), 9

`is_gzipped` (pynion.filesystem._filetypes.ContainerFile attribute), 9

`is_open` (pynion.filesystem._filetypes.BaseFile attribute), 8

`is_tarfile` (pynion.filesystem._filetypes.ContainerFile attribute), 9

`is_to_read` (pynion.filesystem._filetypes.BaseFile attribute), 8

`is_to_write` (pynion.filesystem._filetypes.BaseFile attribute), 8

is_verbose() (pynion.Manager method), 5
is_zipped (pynion.filesystem._filetypes.ContainerFile attribute), 9

J

JSONer (class in pynion), 14

L

last_dir (pynion.filesystem._filetypes.BaseFile attribute), 7
length() (pynion.filesystem._filetypes.ContainerFile method), 10
list_directories() (pynion.Path method), 11
list_files() (pynion.filesystem._filetypes.ContainerFile method), 10
list_files() (pynion.Path method), 11

M

Manager (class in pynion), 3
mkdir() (pynion.Path method), 11
Multiton (class in pynion), 14

N

name (pynion.filesystem._filetypes.BaseFile attribute), 7
name (pynion.Path attribute), 11

O

open() (pynion.filesystem._filetypes.BaseFile method), 8
open() (pynion.filesystem._filetypes.CompressedFile method), 9
open() (pynion.filesystem._filetypes.ContainerFile method), 10

P

parent (pynion.Path attribute), 10
parents (pynion.Path attribute), 10
Path (class in pynion), 10
path (pynion.Executable attribute), 12
pattern (pynion.filesystem._filetypes.BaseFile attribute), 8
prefix (pynion.filesystem._filetypes.BaseFile attribute), 7
pynion (module), 1
pynion.abstractclass (module), 14
pynion.metaclass (module), 13

R

read() (pynion.filesystem._filetypes.BaseFile method), 8
read_file() (pynion.filesystem._filetypes.ContainerFile method), 10
readJSON() (pynion.filesystem._filetypes.BaseFile method), 8
readline() (pynion.filesystem._filetypes.BaseFile method), 8

relative_to() (pynion.filesystem._filetypes.BaseFile method), 8
relative_to() (pynion.Path method), 11
restore_command() (pynion.Executable method), 13

S

set_debug() (pynion.Manager method), 4
set_detail() (pynion.Manager method), 4
set_logfile() (pynion.Manager method), 5
set_overwrite() (pynion.Manager method), 5
set_stdout() (pynion.Manager method), 4
set_unclean() (pynion.Manager method), 4
set_verbose() (pynion.Manager method), 4
Singleton (class in pynion), 13
size (pynion.filesystem._filetypes.BaseFile attribute), 7
sync_from() (pynion.Path method), 12
sync_to() (pynion.Path method), 12

T

to_dict() (pynion.JSONer method), 15
to_json() (pynion.JSONer method), 14

W

warning() (pynion.Manager method), 6
write() (pynion.filesystem._filetypes.BaseFile method), 8