pyworkdir Documentation

pyworkdir

Sep 20, 2019

CONTENTS

1 Basic usage **3**

2 Directories are Customizable Classes **5**

3 Directories have a Command Line Interface **7**

4 Changing Environment Variables **9**

5 Yaml Files **11**

6 Logging **13**

7 API **15**

7.1 workdir 15

7.2 util 19

7.3 main 20

8 Indices and tables **21**

Index **23**

Python Working Directories

Visit project home on GitHub.

CHAPTER

ONE

BASIC USAGE

Changing the current working directory:

```
from pyworkdir import WorkDir

with WorkDir("some_directory"):
    # everything in this context is run
    # in the specified directory
    pass
```

CHAPTER

TWO

DIRECTORIES ARE CUSTOMIZABLE CLASSES

WorkDir classes can be be customized by adding a file *workdir.py* to the directory. All variables, functions, or classes defined in this file will be added as attributes of the *WorkDir* instances.

For instance, consider the following *workdir.py* file:

```
# -- workdir.py --
def data_file(workdir, filename="data.csv"):
    return workdir/filename
```

The function can now be accessed from other code as follows:

```
from pyworkdir import WorkDir

with WorkDir() as wd:
    print(wd.data_file())
```

Note that the parameter *workdir* behaves like the *self* argument of the method. If *workdir* is not an argument of the function, the function behaves like a static method.

By default, the *WorkDir* instance also recursively inherits attributes defined in its parent directory's *workdir.py* files. Therefore, subdirectories behave like subclasses.

CHAPTER

THREE

DIRECTORIES HAVE A COMMAND LINE INTERFACE

Custom functions of the *WorkDir* are directly accessible from a terminal via the command *workdir*. Before being called from the command line, all function parameters (except the reserved keywords *workdir* and *here*) have to be declared as Click options:

```
# -- workdir.py --
import click

num_apples = 2

@click.option("-c", type=int, default=12, help="A number (default:12)")
@click.option("-s","--somebody", type=str, help="A name")
def hello(count, somebody, workdir):
    """This function says hello."""
    workdir.num_apples += 1
    print(
        f"{count} times Hello! to {somebody}: "
        f"we have {workdir.num_apples} apples."
    )
```

Calling the function from the command line looks like this:

```
foo@bar:~$  workdir hello --help
Usage: workdir hello [OPTIONS]

  This function says hello.

Options:
  -c, --count INTEGER  A number (default:12)
  -s, --somebody TEXT  A name
  --help               Show this message and exit.


foo@bar:~$ workdir hello -s "you"
12 times Hello! to you: we have 3 apples.
```

Writing *workdir.py* files like this makes it easy to define local functions that can be called both from inside python and from a terminal. For the latter, the *workdir.py* behaves similar to a Makefile.

To suppress generation of the command line interface for a function, pyworkdir provides a *no_cli* decorator:

```
# -- workdir.py --

from pyworkdir import no_cli

```

(continues on next page)

(continued from previous page)

```
@no_cli
def a_function_without_command_line_interface():
    pass
```

CHAPTER

FOUR

CHANGING ENVIRONMENT VARIABLES

Temporary changes of the environment:

```
from pyworkdir import WorkDir

with WorkDir(environment={"MY_ENVIRONMENT_VARIABLE":"1"}):
    # in this context the environment variable is set
    pass

# outside the context, it is not set any longer
```

CHAPTER

FIVE

YAML FILES

Environment variables and simple attributes can also be set through yml files. The templates *{{ workdir }}* and *{{ here }}* are available and will be replaced by the working directory instance and the directory that contains the yml file, respectively:

```
# -- workdir.yml --
environment:
    VAR_ONE: "a"
attributes:
    my_number: 1
    my_list:
        - 1
        - 2
        - 3
    my_tmpdir: {{ here/"tmpdir" }}
    my_local_tmpfile: {{ workdir/"file.tmp" }}
commands:
    echo: echo Hello // print Hello to the command line
```

The commands are shortcuts for terminal commands that can be called from python and from the command line. Everything after *//* is used as a documentation string for the command line interface. The attributes and environment variables get added to the WorkDir:

```
import os

with WorkDir() as wd:
    print(wd.my_number + 5, wd.my_tmpdir , wd.my_local_tmpfile)
    for el in wd.my_list:
         print(el)
    print(os.environ["VAR_ONE"])
```

Note that environment variables passed to the constructor have preference over those in a yml file.

CHAPTER

SIX

LOGGING

A logger is available:

```
from pyworkdir import WorkDir
import logging

wd = WorkDir()
wd.log("a INFO-level message")
wd.log("a DEBUG-level message", logging.DEBUG)
```

By default, INFO-level and higher is printed to the console. DEBUG-level output is only printed to a file *workdir.log*.

CHAPTER

SEVEN

API

7.1 workdir

Python working directories.

class pyworkdir.workdir.**WorkDir**(*directory='.', mkdir=True, python_files=['workdir.py'], yml_files=['workdir.yml'], python_files_recursion=-1, yml_files_recursion=-1, environment={}, logger=None, logfile='workdir.log', loglevel_console=20, loglevel_file=10*)

Bases: object

Working directory class.

Parameters

- **directory** (*str, Optional, default: "."*) – The directory name
- **mkdir** (*bool, Optional, default: True*) – Whether to create the directory if it does not exist
- **python_files** (*list of string, Optional, default: ["workdir.py"]*) – A list of python files. All variables, functions, and classes defined in these files are added as members to customize the WorkDir.
- **yml_files** (*list of string, Optional, default: ["workdir.yml"]*) – A list of configuration files to read a configuration from.
- **python_files_recursion** (*int, Optional, default: -1*) – Recursion level for loading python files from parent directories. 0 means only this directory, 1 means this directory and its parent directory, etc. If -1, recurse until root.
- **yml_files_recursion** (*int, Optional, default: -1*) – Recursion level for yml files.
- **environment** (*dict, Optional, default: dict()*) – A dictionary. Keys (names of environment variables) and values (values of environment variables) have to be strings. Environment variables are temporarily set to these values within a context (a *with WorkDir() ...* block) and set to their original values outside the context.
- **logger** (*logging.Logger or None, Optional, default: None*) – A logger instance. If None, use a default logger. If a custom logger is specified, the other arguments that concern the logger are not recognized.
- **logfile** (*str, Optional, default: "workdir.log"*) – The logfile to write output to.
- **loglevel_console** (*int, Optional, default: logging.INFO*) – The level of logging to the console.

- **loglevel_file** (*int, Optional, default: logging.DEBUG*) – The level of logging to the logfile.

path
Absolute path of this working directory

Type pathlib.Path

scope_path
The path of the surrounding scope (when used as a context manager)

Type pathlib.Path

environment
A dictionary of environment variables to be set in the context

Type dict

scope_environment
A dictionary to keep track of the environment of the scope

Type dict

custom_attributes
A dictionary that lists custom attributes of this working directory. The values of the dictionary are the source files which contain the definition of each attribute.

Type dict

python_files
A list of python filenames that the workdir instance may read its custom attributes from. Files do not need to exist.

Type list of str

yml_files
A list of yml filenames that the workdir instance may read its custom attributes from. Files do not need to exist.

Type list of str

logger
A logger instance

Type logging.Logger or None

logfile
Filename of the log file

Type str

loglevel_console
An integer between 0 (logging.NOT_SET) and 50 (logging.CRITICAL) for level of printing to the console

Type int

loglevel_file
An integer between 0 (logging.NOT_SET) and 50 (logging.CRITICAL) for level of printing to the file

Type int

commands
A dictionary of terminal commands.

Type dict

Notes

Get the absolute path of a file in this working directory

```
>>> with WorkDir("some_path") as wd:
>>>     absolute_path = wd / "some_file.txt"
```

Get the number of files and subdirectories:

```
>>>     len(wd)
```

Iterate over all files in this working directory:

```
>>>     for f in wd.files():
>>>         pass
```

Examples

Basic usage:

```
>>> with WorkDir("some_path"):
>>>     # everything in this context will
>>>     # run in the specified directory
>>>     pass
```

Customizing the working directory:

To add or change attributes of the WorkDir, create a file "workdir.py" in the directory. All functions, classes, and variables defined in "workdir.py" will be added as attributes to the WorkDir.

```
>>> # -- workdir.py --
>>> def custom_sum_function(a, b):
>>>     return a + b
```

```
>>> # -- main.py --
>>> wd = WorkDir(".")
>>> result = wd.custom_sum_function(a,b)
```

By default, these attributes get added recursively from parent directories as well, where more specific settings (further down in the directory tree) override more general ones. This mimics a kind of inheritance, where subdirectories inherit attributes from their parents.

When defining functions in the workdir.py file, some argument names have special meaning: - The argument name *workdir* refers to the working directory instance.

    It represents the *self* argument of the method.

- The argument name *here* refers to the absolute path of the directory that contains the workdir.py file.

Environment variables can be changed inside a context as follows.

```
>>> import os
>>> with WorkDir(environment={"VAR_ONE": "ONE", "VAR_TWO": "TWO"}):
>>>     print(os.environ["VAR_ONE"])
>>> assert "VAR_ONE" not in os.environ
```

Environment variables and simple attributes can also be set through yml files. The templates *{{ workdir }}* and *{{ here }}* are available and will be replaced by the working directory instance and the directory that contains the yml file, respectively.

```
>>> # -- workdir.yml --
    environment:
        VAR_ONE: "a"
    attributes:
        my_number: 1
        my_list:
            - 1
            - 2
            - 3
        my_tmpdir: {{ here/"tmpdir" }}
        my_local_tmpfile: {{ workdir/"file.tmp" }}
```

```
>>> with WorkDir() as wd:
>>>     print(wd.my_number + 5, wd.my_tmpdir , wd.my_local_tmpfile)
>>>     for el in wd.my_list:
>>>         print(el)
>>>     print(os.environ["VAR_ONE"])
```

Note that environment variables passed to the constructor have preference over those in a yml file.

A logging instance is available; the default output file is workdir.log:

```
>>> wd = WorkDir()
>>> wd.log("my message")
>>> import logging
>>> wd.log("debug info", level=logging.DEBUG)
```

add_members_from_pyfile(*pyfile*)
Initialize members of this WorkDir from a python file.

The following attributes are not added as members of the WorkDir:

1) imported modules
2) built-ins and private objects, i.e. if the name starts with an underscore
3) objects that are imported from other modules using *from ... import ...*

The only exception to 3. is if the imported function has a command-line interface, i.e. *@click.option*-decorated functions added to the workdir so that they can be called from the command line.

Parameters **pyfile** (*path-like object*) – Absolute path of a python file.

Notes

The function arguments *workdir* and *here* of imported functions are replaced by the WorkDir instance and the directory containing the pyfile, respectively.

add_members_from_yml_file(*yml_file*)
Initialize members and environment variables from a yml file.

files(*abs=False*)
Iterator over files in this work dir.

Parameters **abs** (*bool, Optional, default=False*) – Yield absolute filenames

Yields **file** (*str*) – Filenames in this directory

Examples

```
>>> with WorkDir("some_directory") as wd:
>>>     for file in wd.files():
>>>         print(file)
```

log(*message*, *level=20*)
Write logging output to the console and/or a log file.

Parameters

- **message** (*str*) –
- **level** (*int, Optional, default: logging.DEBUG*) –

7.2 util

Utilities for workdir

exception pyworkdir.util.**WorkDirException**
Bases: Exception

General exception class for pyworkdir module.

pyworkdir.util.**forge_method**(*instance*, *func*, *replace_args={}*, *name=None*, *add=True*)
Forge a method and add it to an instance.

Parameters

- **instance** (*class instance*) – The instance to which the function should be added as a method
- **func** (*function*) – The function to be added to the instance
- **replace_args** (*dict, Optional, default = dict()*) – Any arguments that are replaced by default values in the spirit of functools.partial
- **name** (*str, Optional, default=None*) – The function's name; if None, infer from function.__name__
- **add** (*bool, Optional, default=True*) – If False, do not add the function but return it instead.

Notes

This function takes care of option-decorated functions. They retain their __click_params__ field; also all *replace_args* get added as hidden options so that they are not visible on the command line interface.

pyworkdir.util.**import_from_file**(*filename*)
Import a python module from a file by path.

Parameters **filename** (*str or path-like*) – The file to be imported

Returns **pymod** – The imported module

Return type python module

pyworkdir.util.**recursively_get_filenames**(*path*, *filenames*, *recursion_depth*, *current_recursion_level=0*)
Get all filenames (python/yaml) that attributes should be read from.

Parameters

- **path** (*str or path-like object*) – the current directory
- **filenames** (*str*) – The base filenames.
- **recursion_depth** (*int*) – The maximum recursion depth (0 = only current directory, 1 = current and parents). -1 means recurse until root.
- **current_recursion_level** (*int, Optional, default = 0*) – Current recursion level of the function.

Returns **filenames** – A list of filenames, where the ones further up front in the list are further up in the directory tree. The files do not need to exist.

Return type list

7.3 main

Command line interface

pyworkdir.main.**bash_function**(*bash_command*)
Bash command as a python function

Parameters **bash_command** (*str*) –

Returns **function** – A python function that runs the bash command.

Return type callable

pyworkdir.main.**entrypoint**()
Entrypoint for the workdir command.

pyworkdir.main.**forge_command_line_interface**(**args*, ***kwargs*)
Forge the click.Group that holds all commands defined in workdir.py All arguments are forwarded to the constructor of WorkDir.

Returns

Return type A click.Group that defines one command for every custom function in the WorkDir.

pyworkdir.main.**no_cli**(*function*)
Function decorator to suppress generation of a command-line interface for this function.

Examples

```
>>> # in workdir.py
>>> from pyworkdir import no_cli
>>>
>>> @no_cli
>>> def function_without_command_line_interace()
>>>     pass
```

pyworkdir.main.**show**(*workdir*, *variables=False*, *functions=False*, *sources=False*, *environment=False*, *commands=False*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Print the working directory in yaml format.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C

commands (*pyworkdir.workdir.WorkDir attribute*), 16
custom_attributes (*pyworkdir.workdir.WorkDir attribute*), 16

E

environment (*pyworkdir.workdir.WorkDir attribute*), 16

L

logfile (*pyworkdir.workdir.WorkDir attribute*), 16
logger (*pyworkdir.workdir.WorkDir attribute*), 16
loglevel_console (*pyworkdir.workdir.WorkDir attribute*), 16
loglevel_file (*pyworkdir.workdir.WorkDir attribute*), 16

P

path (*pyworkdir.workdir.WorkDir attribute*), 16
python_files (*pyworkdir.workdir.WorkDir attribute*), 16

S

scope_environment (*pyworkdir.workdir.WorkDir attribute*), 16
scope_path (*pyworkdir.workdir.WorkDir attribute*), 16

Y

yml_files (*pyworkdir.workdir.WorkDir attribute*), 16