

---

# **PyWinCFFI Documentation**

***Release 0.1.2***

**Oliver Palmer**

January 22, 2016



<b>1</b>	<b>Main Index</b>	<b>3</b>
1.1	Changelog . . . . .	3
<b>2</b>	<b>Python Package</b>	<b>5</b>
2.1	pywincffi . . . . .	5
<b>3</b>	<b>Development</b>	<b>13</b>
3.1	Development . . . . .	13
	<b>Python Module Index</b>	<b>17</b>



`pywincffi` is a wrapper around some Windows API functions using Python and the `ffi` library. This project was originally created to assist the Twisted project in moving away from its dependency on `pywin32`. Contributions to expand on the APIs which `pywincffi` offers are always welcome however.

The core objectives and design principles behind this project are:

- It should be easier to use Windows API functions both in terms of implementation and distribution.
- Python 2.6, 2.7 and 3.x should be supported from a single code base and not require a consumer of `pywincffi` to worry about how they use the library.
- Type conversion, error checking and other ‘C like’ code should be the responsibility of the library where possible.
- APIs provided by `pywincffi` should mirror their Windows counterparts as closely as possible so the MSDN documentation can be more easily used as reference.
- Documentation and error messages should be descriptive, consistent, complete and accessible. Examples should be provided for more complex use cases.
- For contributors, it should be possible to develop and test regardless of what platform the contributor is coming from.

**See also:**

[PyWinCFFI's README](#)



## 1.1 Changelog

This document contains information on pywinffi's release history. Later versions are shown first.

### 1.1.1 Versions

#### 0.1.2

Contains a fix to ensure that the proper version of `ffi` is installed. See <https://github.com/opalmer/pywinffi/pull/45> for more detailed information. This release also includes a fix to the internal release tool.

#### 0.1.1

The first public release of pywinffi. The [GitHub release](#) contains the full list of issues, changes and pull requests. The primary purpose of this release was to end up with the tools and code necessary to begin integrating pywinffi into Twisted.

#### 0.1.0

This was an internal test release. No data was published to PyPi or GitHub.

- [genindex](#)
- [modindex](#)
- [search](#)





---

## Python Package

---

### 2.1 pywincffi

#### 2.1.1 pywincffi package

##### Subpackages

`pywincffi.core` package

##### Submodules

##### `pywincffi.core.checks` module

**Checks** Provides functions that are responsible for internal type checks.

**class** `pywincffi.core.checks.CheckMapping` (*kind, cname, nullable*)  
Bases: `tuple`

**cname**  
Alias for field number 1

**kind**  
Alias for field number 0

**nullable**  
Alias for field number 2

**class** `pywincffi.core.checks.Enums`  
Bases: `enum.Enum`

**HANDLE** = `<Enums.HANDLE: 2>`

**NON\_ZERO** = `<Enums.NON_ZERO: 1>`

**OVERLAPPED** = `<Enums.OVERLAPPED: 4>`

**PYFILE** = `<Enums.PYFILE: 5>`

**UTF8** = `<Enums.UTF8: 3>`

`pywincffi.core.checks.error_check` (*api\_function, code=None, expected=0*)

Checks the results of a return code against an expected result. If a code is not provided we'll use `ffi.getwinerror()` to retrieve the code.

### Parameters

- **api\_function** (*str*) – The Windows API function being called.
- **code** (*int*) – An explicit code to compare against. This can be used instead of asking `ffi.getwinerrro()` to retrieve a code.
- **expected** (*int*) – The code we expect to have as a result of a successful call. This can also be passed `pywincffi.core.checks.Enums.NON_ZERO` if code can be anything but zero.

**Raises** *pywincffi.exceptions.WindowsAPIError* Raised if we receive an unexpected result from a Windows API call

`pywincffi.core.checks.input_check(name, value, allowed_types=None, allowed_values=None)`

A small wrapper around `isinstance()`. This is mainly meant to be used inside of other functions to pre-validate input rather than using assertions. It's better to fail early with bad input so more reasonable error message can be provided instead of from somewhere deep in cffi or Windows.

### Parameters

- **name** (*str*) – The name of the input being checked. This is provided so error messages make more sense and can be attributed to specific input arguments.
- **value** – The value we're performing the type check on.
- **allowed\_types** – The allowed type or types for value. This argument also supports a special value, `pywincffi.core.checks.Enums.HANDLE`, which will check to ensure value is a handle object.
- **allowed\_values** (*tuple*) – A tuple of allowed values. When provided value must be in this tuple otherwise `InputError` will be raised.

**Raises** *pywincffi.exceptions.InputError* Raised if value is not an instance of allowed\_types

## pywincffi.core.config module

**Configuration** Simple module for loading pywincffi's configuration, intended for internal use. A configuration file which is reasonable for every day use ships with pywincffi but can be overridden at runtime by dropping a file named `pywincffi.ini` in the current working directory or the current users's home directory.

**class** `pywincffi.core.config.Configuration`

Bases: `ConfigParser.RawConfigParser`

Class responsible for loading and retrieving data from the configuration files. This is used by a few parts of pywincffi to control various aspects of execution.

**FILES** = (`'/home/docs/checkouts/readthedocs.org/user_builds/pywincffi/checkouts/0.1.2/pywincffi/core/pywincffi.ini'`, `'/ho`

**LOGGER\_LEVEL\_MAPPINGS** = `{'info': 20, 'warning': 30, 'critical': 50, 'error': 40, 'debug': 10, 'notset': 0}`

**load()**

Loads the configuration from disk

**logging\_level()**

Returns the logging level that the configuration currently dictates.

## pywincffi.core.dist module

**Distribution** Module responsible for building the pywinffi distribution in `setup.py`. This module is meant to serve two purposes. The first is to serve as the main means of loading the pywinffi library:

```
>>> from pywinffi.core import dist
>>> ffi, lib = dist.load()
```

The second is to facilitate a means of building a static library. This is used by the `setup.py` during the install process to build and install pywinffi as well as a wheel for distribution.

```
pywinffi.core.dist.load()
```

The main function used by pywinffi to load an instance of FFI and the underlying build library.

## pywinffi.core.logger module

**Logger** This module contains pywinffi's logger and provides functions to configure the logger at runtime.

```
pywinffi.core.logger.get_logger(name)
```

Returns an instance of `logging.Logger` as a child of pywinffi's main logger.

**Parameters** `name` (*str*) – The name of the child logger to return. For example, if you provide *foo* for the name the resulting name will be *pywinffi.foo*.

**Raises ValueError** Raised if `name` starts with a dot.

**Return type** `logging.Logger`

## Module contents

**Core Sub-Package** An internal package used by pywinffi for loading the underlying `_pywinffi` module, handling configuration data, logging and other common tasks. This package also contains the C source and header files.

## pywinffi.dev package

### Submodules

#### pywinffi.dev.lint module

**Lint Utilities** Provides some help to pylint so static analysis can be made aware of some constants and functions that we define in headers.

```
pywinffi.dev.lint.constants_in_file(path)
```

Returns a set of constants in the given file path

```
pywinffi.dev.lint.functions_in_file(path)
```

Returns a set of functions defined in the given file path

```
pywinffi.dev.lint.register(_)
```

An entypoint that pylint uses to search for and register plugins with the given `linter`

```
pywinffi.dev.lint.transform(cls, constants=None, functions=None)
```

Transforms class objects from pylint so they're aware of extra attributes that are not present when being statically analyzed.

#### pywinffi.dev.release module

## pywincffi.dev.testutil module

**Test Utility** This module is used by the unittests.

**class** `pywincffi.dev.testutil.TestCase` (*methodName='runTest'*)

Bases: `unittest.case.TestCase`

A base class for all test cases. By default the core test case just provides some extra functionality.

**SetLastError** (*value=0, lib=None*)

Calls the Windows API function SetLastError()

**setUp** ()

## Module contents

**Development Sub-Package** This package is used for development, testing and release purposes. It does not contain core functionality of pywincffi and is unused by `pywincffi.core`, `pywincffi.kernel32` and other similar modules.

## pywincffi.kernel32 package

### Submodules

## pywincffi.kernel32.io module

**Files** A module containing common Windows file functions.

`pywincffi.kernel32.io.CloseHandle` (*hObject*)

Closes an open object handle.

**See also:**

<https://msdn.microsoft.com/en-us/library/ms724211>

**Parameters** `hObject` (*handle*) – The handle object to close.

`pywincffi.kernel32.io.CreatePipe` (*nSize=0, lpPipeAttributes=None*)

Creates an anonymous pipe and returns the read and write handles.

**See also:**

<https://msdn.microsoft.com/en-us/library/aa365152> <https://msdn.microsoft.com/en-us/library/aa379560>

```
>>> from pywincffi.core import dist
>>> ffi, library = dist.load()
>>> lpPipeAttributes = ffi.new(
...     "SECURITY_ATTRIBUTES[1]", [{
...         "nLength": ffi.sizeof("SECURITY_ATTRIBUTES"),
...         "bInheritHandle": True,
...         "lpSecurityDescriptor": ffi.NULL
...     }]
... )
>>> reader, writer = CreatePipe(lpPipeAttributes=lpPipeAttributes)
```

**Parameters**

- **nSize** (*int*) – The size of the buffer in bytes. Passing in 0, which is the default will cause the system to use the default buffer size.
- **lpPipeAttributes** – The security attributes to apply to the handle. By default `NULL` will be passed in meaning then handle we create cannot be inherited. For more detailed information see the links below.

**Returns** Returns a tuple of handles containing the reader and writer ends of the pipe that was created. The user of this function is responsible for calling `CloseHandle` at some point.

`pywincffi.kernel32.io.GetStdHandle (nStdHandle)`

Retrieves a handle to the specified standard device (standard input, standard output, or standard error).

**See also:**

<https://msdn.microsoft.com/en-us/library/ms683231>

**Parameters** **nStdHandle** (*int*) – The standard device to retrieve

**Return type** handle

**Returns** Returns a handle to the standard device retrieved.

`pywincffi.kernel32.io.PeekNamedPipe (hNamedPipe, nBufferSize)`

Copies data from a pipe into a buffer without removing it from the pipe.

**See also:**

<https://msdn.microsoft.com/en-us/library/aa365779>

**Parameters**

- **hNamedPipe** (*handle*) – The handle to the pipe object we want to peek into.
- **nBufferSize** (*int*) – The number of bytes to ‘peek’ into the pipe.

**Return type** *PeekNamedPipeResult*

**Returns** Returns an instance of *PeekNamedPipeResult* which contains the buffer read, number of bytes read and the result.

`class pywincffi.kernel32.io.PeekNamedPipeResult (lpBuffer, lpBytesRead, lpTotalBytesAvail, lpBytesLeftThisMessage)`

Bases: tuple

**lpBuffer**

Alias for field number 0

**lpBytesLeftThisMessage**

Alias for field number 3

**lpBytesRead**

Alias for field number 1

**lpTotalBytesAvail**

Alias for field number 2

`pywincffi.kernel32.io.ReadFile (hFile, nNumberOfBytesToRead, lpOverlapped=None)`

Read the specified number of bytes from `hFile`.

**See also:**

<https://msdn.microsoft.com/en-us/library/aa365467>

#### Parameters

- **hFile** (*handle*) – The handle to read from
- **nNumberOfBytesToRead** (*int*) – The number of bytes to read from hFile
- **lpOverlapped** (*None or OVERLAPPED*) – None or a pointer to a OVERLAPPED structure. See Microsoft's documentation for intended usage and below for an example of this struct.

```
>>> from pywincffi.core import dist
>>> ffi, library = dist.load()
>>> hFile = None # normally, this would be a handle
>>> lpOverlapped = ffi.new(
...     "OVERLAPPED[1]", [{
...         "hEvent": hFile
...     }]
... )
>>> read_data = ReadFile( # read 12 bytes from hFile
...     hFile, 12, lpOverlapped=lpOverlapped)
```

**Returns** Returns the data read from hFile

`pywincffi.kernel32.io.SetNamedPipeHandleState` (*hNamedPipe, lpMode=None, lpMaxCollectionCount=None, lpCollectDataTimeout=None*)

Sets the read and blocking mode of the specified hNamedPipe.

**See also:**

<https://msdn.microsoft.com/en-us/library/aa365787>

#### Parameters

- **hNamedPipe** (*handle*) – A handle to the named pipe instance.
- **lpMode** (*int*) – The new pipe mode which is a combination of read mode:
  - `PIPE_READMODE_BYTE`
  - `PIPE_READMODE_MESSAGE`And a wait-mode flag:
  - `PIPE_WAIT`
  - `PIPE_NOWAIT`
- **lpMaxCollectionCount** (*int*) – The maximum number of bytes collected.
- **lpCollectDataTimeout** (*int*) – The maximum time, in milliseconds, that can pass before a remote named pipe transfers information

`pywincffi.kernel32.io.WriteFile` (*hFile, lpBuffer, lpOverlapped=None*)

Writes data to hFile which may be an I/O device for file.

**See also:**

<https://msdn.microsoft.com/en-us/library/aa365747>

#### Parameters

- **hFile** (*handle*) – The handle to write to
- **lpBuffer** (*bytes, string or unicode.*) – The data to be written to the file or device. We should be able to convert this value to unicode.
- **lpOverlapped** (*None or OVERLAPPED*) – None or a pointer to a OVERLAPPED structure. See Microsoft's documentation for intended usage and below for an example of this struct.

```
>>> from pywinffi.core import dist
>>> ffi, library = dist.load()
>>> hFile = None # normally, this would be a handle
>>> lpOverlapped = ffi.new(
...     "OVERLAPPED[1]", [{
...         "hEvent": hFile
...     }]
... )
>>> bytes_written = WriteFile(
...     hFile, "Hello world", lpOverlapped=lpOverlapped)
```

**Returns** Returns the number of bytes written

`pywinffi.kernel32.io.handle_from_file` (*python\_file*)

Given a standard Python file object produce a Windows handle object that be be used in Windows function calls.

**Parameters** `python_file` (*file*) – The Python file object to convert to a Windows handle.

**Returns** Returns a Windows handle object which is pointing at the provided `python_file` object.

## pywinffi.kernel32.process module

**Process** Provides functions, constants and utilities that wrap the Windows functions associated with process management and interaction. This module also provides several constants as well, see Microsoft's documentation for the constant names and their purpose:

- **Process Security and Access Rights** - <https://msdn.microsoft.com/en-us/library/windows/desktop/ms684880>

---

**Note:** Not all constants may be defined

---

`pywinffi.kernel32.process.OpenProcess` (*dwDesiredAccess, bInheritHandle, dwProcessId*)

Opens an existing local process object.

**See also:**

<https://msdn.microsoft.com/en-us/library/ms684320>

### Parameters

- **dwDesiredAccess** (*int*) – The required access to the process object.
- **bInheritHandle** (*bool*) – Enables or disable handle inheritance for child processes.
- **dwProcessId** (*int*) – The id of the local process to be opened.

**Returns** Returns a handle to the opened process in the form of a void pointer. This value can be used by other functions such as `TerminateProcess()`

## Module contents

**Kernel32 Sub-Package** Provides functions, constants and utilities that wrap functions provided by `kernel32.dll`.

## Submodules

### `pywincffi.exceptions` module

#### Exceptions

Custom exceptions that `pywincffi` can throw.

**exception** `pywincffi.exceptions.ConfigurationError`

Bases: `pywincffi.exceptions.PyWinCFFIError`

Raised when there was a problem with the configuration file

**exception** `pywincffi.exceptions.InputError` (*name*, *value*, *expected\_types*, *allowed\_values=None*, *ffi=None*)

Bases: `pywincffi.exceptions.PyWinCFFIError`

A subclass of `PyWinCFFIError` that's raised when invalid input is provided to a function. Because we're passing inputs to C we have to be sure that the input(s) being provided are what we're expecting so we fail early and provide better error messages.

**exception** `pywincffi.exceptions.PyWinCFFIError`

Bases: `exceptions.Exception`

The base class for all custom exceptions that `pywincffi` can throw.

**exception** `pywincffi.exceptions.ResourceNotFoundError`

Bases: `pywincffi.exceptions.PyWinCFFIError`

Raised when we fail to locate a specific resource

**exception** `pywincffi.exceptions.WindowsAPIError` (*api\_function*, *api\_error\_message*, *code*, *expected\_code*)

Bases: `pywincffi.exceptions.PyWinCFFIError`

A subclass of `PyWinCFFIError` that's raised when there was a problem calling a Windows API function.

## Module contents

### `PyWinCFFI`

The root of the `pywincffi` package. See the README and documentation for help and examples.



---

## Development

---

### 3.1 Development

The documents outlined here cover topics related to development.

#### 3.1.1 Vagrant

Vagrant is used by the pywinffi project to facilitate testing and development on non-windows platforms. While the project does have continuous integration hooked up to commits and pull requests Vagrant can help with local development. The information below details the general steps needed to get Vagrant up and running.

##### Prerequisites

Before starting, you will need a pieces of software preinstalled on your system:

- **Vagrant** - The software used to launch and provision the virtual machine image.
- **Packer** - Used to build a virtual machine image, referred to as a box, which Vagrant can then use.

##### Building The Base Virtual Machine Image

In order to effectively run and test pywinffi you must have access to a Windows host, various versions of Python and a couple of different compilers. While you can rely on continuous integration to provide this it's faster to test locally usually.

The install process for the various dependencies besides the operating system will be covered in another section. This section will cover setting up the base machine image itself.

1. Use git to clone the packer templates:

```
git clone https://github.com/mwrock/packer-templates.git
```

This repository contains all the code necessary to build our base image. For some extra information on how this works you can see this article:

<http://www.hurryupandwait.io/blog/creating-windows-base-images-for-virtualbox-and-hyper-v-using-packer-boxstarter-and-vagrant>

2. Run packer. This will generate the box image which vagrant will need to spin up a virtual machine.

```
cd packer-templates
packer build -force -only virtualbox-iso ./vbox-2012r2.json
```

The above will take a while to run. When complete you should end up with a file like `windows2012r2min-virtualbox.box` on disk.

3. Add the box image to vagrant:

```
vagrant box add windows2012r2min-virtualbox.box --name windows2012r2min
```

At this point, you should have everything you need to launch vagrant with a Windows image.

---

**Note:** The box that was generated is using an evaluation copy of Windows 2012 R2 Standard which expires in 180 days. You will either need to add a license for the operating system or repeat the steps outlined above again later on.

---

### Running Vagrant

Vagrant is responsible for running the virtual machine as well as installing and downloading the necessary software for pywincffi. The process for launching vagrant is:

```
cd <path to clone of pywincffi>
vagrant up --provider virtualbox
```

This will start up the virtual machine, download the necessary software and get it installed on the system.

---

**Important:** At certain points during the install you will be required to perform some manual steps. This is because certain software, such as Visual Studio express editions, can't easily be installed in an unattended manner.

---

### Rerunning The Provisioning Step

Sometimes you might need to execute the provisioning process again. This could be because one of the steps failed when running `vagrant up`, you've added a new step to the Vagrantfile or you've modified a step in `.ci/vagrant/`.

To reexecute the provisioning process on a running VM run:

```
vagrant provision
```

To restart the VM and execute the provisioning process run:

```
vagrant reload --provision
```

### Installing Python Source Code

By default, going back over *rerunning the provisioning step* will install the source code for you. If you make changes however to the `setup.py` file or something seems broken you can force the provision process to run again and skip the OS steps:

```
vagrant provision --provision-with python,install
```

## Adding SSH Authorized Keys

SSH for the Windows VM is setup to use key based authentication. To provide you own set of keys, create a file at `.ci/vagrant/files/authorized_keys` with your own public key(s).

pywincffi ships `.ci/vagrant/files/authorized_keys.template` which contains vagrant's public key. You're welcome to copy this over and add your own keys. By doing this, you'll be able to run `vagrant ssh` in addition to being able to use ssh directly with your own key.

In addition you can also use the `vagrant` password for either the `vagrant` account or the Administrator account to login manually if needed.

## Testing PyWinCFFI

### PyCharm Remote Interpreter

If you're using [PyCharm](#) you can take advantage of its remote interpreter feature. This will allow you to execute the tests as if Python is running locally even though it's in a virtual machine.

For more information on how to set this up, check out these guides direct from JetBrains:

- <https://www.jetbrains.com/pycharm/help/configuring-remote-python-interpreters.html>
- <https://confluence.jetbrains.com/display/PYH/Configuring+Interpreters+with+PyCharm>
- <https://www.jetbrains.com/pycharm/help/configuring-remote-interpreters-via-virtual-boxes.html>

---

**Note:** Some of the features above may require the professional version of PyCharm.

---

## Manually Testing Using Vagrant

**Warning:** This method of testing does not work currently. Please use one of these methods instead:

- *PyCharm Remote Interpreter*
- *Manually Using SSH and CYGWIN*

**Issue:** <https://github.com/opalmer/pywincffi/issues/28>

Before attempting to test be sure the core Python interpreters have been installed:

```
vagrant provision --provision-with python,install
```

If you add a new module or the tests seem to be failing due to recent project changes you can rerun the above steps.

Next, execute the tests:

```
vagrant provision --provision-with test
```

## Manually Using SSH and CYGWIN

You can also manually test the project as well over ssh.

```
$ ssh -p 2244 vagrant@localhost
$ cd /cygdrive/c/code
$ ~/virtualenv/2.7.10-x86/Scripts/python.exe setup.py test
[ ... ]
```

```
-----  
Ran 70 tests in 0.359s
```

```
OK
```

## p

- `pywincffi`, [12](#)
- `pywincffi.core`, [7](#)
- `pywincffi.core.checks`, [5](#)
- `pywincffi.core.config`, [6](#)
- `pywincffi.core.dist`, [6](#)
- `pywincffi.core.logger`, [7](#)
- `pywincffi.dev`, [8](#)
- `pywincffi.dev.lint`, [7](#)
- `pywincffi.dev.testutil`, [8](#)
- `pywincffi.exceptions`, [12](#)
- `pywincffi.kernel32`, [11](#)
- `pywincffi.kernel32.io`, [8](#)
- `pywincffi.kernel32.process`, [11](#)



## C

CheckMapping (class in pywincffi.core.checks), 5  
CloseHandle() (in module pywincffi.kernel32.io), 8  
cname (pywincffi.core.checks.CheckMapping attribute), 5  
Configuration (class in pywincffi.core.config), 6  
ConfigurationError, 12  
constants\_in\_file() (in module pywincffi.dev.lint), 7  
CreatePipe() (in module pywincffi.kernel32.io), 8

## E

Enums (class in pywincffi.core.checks), 5  
error\_check() (in module pywincffi.core.checks), 5

## F

FILES (pywincffi.core.config.Configuration attribute), 6  
functions\_in\_file() (in module pywincffi.dev.lint), 7

## G

get\_logger() (in module pywincffi.core.logger), 7  
GetStdHandle() (in module pywincffi.kernel32.io), 9

## H

HANDLE (pywincffi.core.checks.Enums attribute), 5  
handle\_from\_file() (in module pywincffi.kernel32.io), 11

## I

input\_check() (in module pywincffi.core.checks), 6  
InputError, 12

## K

kind (pywincffi.core.checks.CheckMapping attribute), 5

## L

load() (in module pywincffi.core.dist), 7  
load() (pywincffi.core.config.Configuration method), 6  
LOGGER\_LEVEL\_MAPPINGS (pywincffi.core.config.Configuration attribute), 6

logging\_level() (pywincffi.core.config.Configuration method), 6  
lpBuffer (pywincffi.kernel32.io.PeekNamedPipeResult attribute), 9  
lpBytesLeftThisMessage (pywincffi.kernel32.io.PeekNamedPipeResult attribute), 9  
lpBytesRead (pywincffi.kernel32.io.PeekNamedPipeResult attribute), 9  
lpTotalBytesAvail (pywincffi.kernel32.io.PeekNamedPipeResult attribute), 9

## N

NON\_ZERO (pywincffi.core.checks.Enums attribute), 5  
nullable (pywincffi.core.checks.CheckMapping attribute), 5

## O

OpenProcess() (in module pywincffi.kernel32.process), 11  
OVERLAPPED (pywincffi.core.checks.Enums attribute), 5

## P

PeekNamedPipe() (in module pywincffi.kernel32.io), 9  
PeekNamedPipeResult (class in pywincffi.kernel32.io), 9  
PYFILE (pywincffi.core.checks.Enums attribute), 5  
pywincffi (module), 12  
pywincffi.core (module), 7  
pywincffi.core.checks (module), 5  
pywincffi.core.config (module), 6  
pywincffi.core.dist (module), 6  
pywincffi.core.logger (module), 7  
pywincffi.dev (module), 8  
pywincffi.dev.lint (module), 7  
pywincffi.dev.testutil (module), 8  
pywincffi.exceptions (module), 12  
pywincffi.kernel32 (module), 11  
pywincffi.kernel32.io (module), 8  
pywincffi.kernel32.process (module), 11  
PyWinCFFIError, 12

## R

`ReadFile()` (in module `pywincffi.kernel32.io`), [9](#)  
`register()` (in module `pywincffi.dev.lint`), [7](#)  
`ResourceNotFoundError`, [12](#)

## S

`SetLastError()` (`pywincffi.dev.testutil.TestCase` method),  
[8](#)  
`SetNamedPipeHandleState()` (in module `py-`  
`wincffi.kernel32.io`), [10](#)  
`setUp()` (`pywincffi.dev.testutil.TestCase` method), [8](#)

## T

`TestCase` (class in `pywincffi.dev.testutil`), [8](#)  
`transform()` (in module `pywincffi.dev.lint`), [7](#)

## U

`UTF8` (`pywincffi.core.checks.Enums` attribute), [5](#)

## W

`WindowsAPIError`, [12](#)  
`WriteFile()` (in module `pywincffi.kernel32.io`), [10](#)