Table des matières

Le registre : winReg

https://openclassrooms.com/forum/sujet/lancer-un-script-python-depuis-un-autre-script-pyt

http://www.pythonforbeginners.com/os/subprocess-for-system-administrators

https://docs.python.org/3/library/winreg.html pour creation / modif ODBC dans registry

https://pabitrap.wordpress.com/2016/09/08/accessing-windows-registry-using-pythons-winreg-module/

http://effbot.org/librarybook/winreg.htm

http://www.programcreek.com/python/index/2100/winreg

https://docs.python.org/3/library/winreg.html

34.3. `winreg` — Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Changed in version 3.3: Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

34.3.1. Functions

This module offers the following functions:

`winreg.CloseKey`(*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

Note

If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

`winreg.ConnectRegistry`(*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a handle object.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Changed in version 3.3: See above.

`winreg.CreateKey`(*key*, *sub_key*)

Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Changed in version 3.3: See above.

`winreg.CreateKeyEx`(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See Access Rights for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

New in version 3.2.

Changed in version 3.3: See above.

`winreg.DeleteKey`(*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

Changed in version 3.3: See above.

`winreg.DeleteKeyEx`(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

Deletes the specified key.

Note

The `DeleteKeyEx()` function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WOW64_64KEY`. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

On unsupported Windows versions, `NotImplementedError` is raised.

New in version 3.2.

Changed in version 3.3: See above.

`winreg.DeleteValue`(*key*, *value*)

Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value is a string that identifies the value to remove.

`winreg.EnumKey`(*key*, *index*)

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined HKEY_* constants.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available.

Changed in version 3.3: See above.

`winreg.EnumValue`(*key*, *index*)

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for `SetValueEx()`)

Changed in version 3.3: See above.

`winreg.ExpandEnvironmentStrings`(*str*)

Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ`:

```
>>>
```

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

winreg.FlushKey(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined HKEY_* constants.

It is not necessary to call FlushKey() to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike CloseKey(), the FlushKey() method returns only when all the data has been written to the registry. An application should only call FlushKey() if it requires absolute certainty that registry changes are on disk.

Note

If you don't know whether a FlushKey() call is required, it probably isn't.

winreg.LoadKey(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by ConnectRegistry() or one of the constants HKEY_USERS or HKEY_LOCAL_MACHINE.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the SaveKey() function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to LoadKey() fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If *key* is a handle returned by ConnectRegistry(), then the path specified in *file_name* is relative to the remote computer.

winreg.OpenKey(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.OpenKeyEx(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

Opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ. See Access Rights for other allowed values.

The result is a new handle to the specified key.

If the function fails, OSError is raised.

Changed in version 3.2: Allow the use of named arguments.

Changed in version 3.3: See above.

winreg.QueryInfoKey(*key*)

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

The result is a tuple of 3 items:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

winreg.QueryValue(*key*, *sub_key*)

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is None or empty, the function retrieves the value set by the SetValue() method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, so always use QueryValueEx() if possible.

winreg.QueryValueEx(*key*, *value_name*)

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for SetValueEx())

winreg.SaveKey(*key*, *file_name*)

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined HKEY_* constants.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the LoadKey() method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for *security_attributes* to the API.

winreg.SetValue(*key*, *sub_key*, *type*, *value*)

Associates a value with a specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the SetValueEx() function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with KEY_SET_VALUE access.

winreg.SetValueEx(*key*, *value_name*, *reserved*, *type*, *value*)

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See Value Types for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

To open the key, use the CreateKey() or OpenKey() methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

`winreg.DisableReflectionKey`(*key*)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise `NotImplemented` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

`winreg.EnableReflectionKey`(*key*)

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise `NotImplemented` if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

`winreg.QueryReflectionKey`(*key*)

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

Returns `True` if reflection is disabled.

Will generally raise `NotImplemented` if executed on a 32-bit operating system.

34.3.2. Constants

The following constants are defined for use in many `_winreg` functions.

34.3.2.1. HKEY_* Constants

`winreg.HKEY_CLASSES_ROOT`

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.HKEY_CURRENT_USER`

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.HKEY_LOCAL_MACHINE`

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.HKEY_USERS`

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.HKEY_PERFORMANCE_DATA`

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

`winreg.HKEY_CURRENT_CONFIG`

Contains information about the current hardware profile of the local computer system.

`winreg.HKEY_DYN_DATA`

This key is not used in versions of Windows after 98.

34.3.2.2. Access Rights

For more information, see Registry Key Security and Access.

`winreg.KEY_ALL_ACCESS`

Combines the STANDARD_RIGHTS_REQUIRED, `KEY_QUERY_VALUE`,

KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights.

winreg.KEY_WRITE

Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY access rights.

winreg.KEY_READ

Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

winreg.KEY_EXECUTE

Equivalent to KEY_READ.

winreg.KEY_QUERY_VALUE

Required to query the values of a registry key.

winreg.KEY_SET_VALUE

Required to create, delete, or set a registry value.

winreg.KEY_CREATE_SUB_KEY

Required to create a subkey of a registry key.

winreg.KEY_ENUMERATE_SUB_KEYS

Required to enumerate the subkeys of a registry key.

winreg.KEY_NOTIFY

Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.KEY_CREATE_LINK

Reserved for system use.

34.3.2.2.1. 64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.KEY_WOW64_64KEY

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

`winreg.KEY_WOW64_32KEY`

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

34.3.2.3. Value Types

For more information, see Registry Value Types.

`winreg.REG_BINARY`

Binary data in any form.

`winreg.REG_DWORD`

32-bit number.

`winreg.REG_DWORD_LITTLE_ENDIAN`

A 32-bit number in little-endian format. Equivalent to `REG_DWORD`.

`winreg.REG_DWORD_BIG_ENDIAN`

A 32-bit number in big-endian format.

`winreg.REG_EXPAND_SZ`

Null-terminated string containing references to environment variables (`%PATH%`).

`winreg.REG_LINK`

A Unicode symbolic link.

`winreg.REG_MULTI_SZ`

A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

`winreg.REG_NONE`

No defined value type.

`winreg.REG_QWORD`

A 64-bit number.

New in version 3.6.

`winreg.REG_QWORD_LITTLE_ENDIAN`

A 64-bit number in little-endian format. Equivalent to REG_QWORD.

New in version 3.6.

winreg.REG_RESOURCE_LIST

A device-driver resource list.

winreg.REG_FULL_RESOURCE_DESCRIPTOR

A hardware setting.

winreg.REG_RESOURCE_REQUIREMENTS_LIST

A hardware resource list.

winreg.REG_SZ

A null-terminated string.

34.3.3. Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the Close() method on the object, or the CloseKey() function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for __bool__() – thus

```
if handle:
    print("Yes")
```

will print Yes if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in int() function), in which case the underlying Windows handle value is returned. You can also use the Detach() method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.Close()

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.Detach()

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

`PyHKEY.__enter__()`
`PyHKEY.__exit__(*exc_info)`

The HKEY object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the `with` block.

Accessing Windows Registry using Python's winreg module

https://pabitrap.wordpress.com/2016/09/08/accessing-windows-registry-using-pythons-winreg-module/

Recently I worked with Python's `_winreg` module (renamed to `winreg` in Python3). The module gives the access to the Windows registry. Using the available methods in the module, one can easily, create, delete keys and values, set given values to specific key, create subkeys etc… Along with, it also defines the registry constants and access privileges for ease of user access.

However, my concerned part was to access the registry key and read their value. Initially, it took me one day to understand how to use this module. Now since, I am pretty much familiar, I am writing this post. But, I understand, the pain of a first time learner.

I used 3 methods. first is `OpenKey` which is used to open a registry key, before you can access it. This method,takes the key that you want to open as the argument. If you want to open a subkey, for a given key, you can pass that as well. Example :-

```
1 from winreg import *
2 ob = OpenKey(HKEY_CURRENT_USER, r'SOFTWARE\Python\PythonCore\3.5')
3 # Here ob is the handle, using which further access (Read) the
```

given subkey.
The detailed syntax is :-

OpenKey(*key*, *sub_key*[, *res*[, *sam*]])

key :- Registry constant (*HKEY_CURRENT_USER* in above example)
sub_key :- particular key you want to access from the given registry constant (‘*SOFTWARE\Python\PythonCore\3.5*‘ in above example)
res :- a reserved integer, and must be zero. The default is zero.
sam :- an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ.

From the syntax, it is clear that, for OpenKey, the default access permission is set to KEY_READ. So, if no access mode is specified, then the key is opened with READ access.

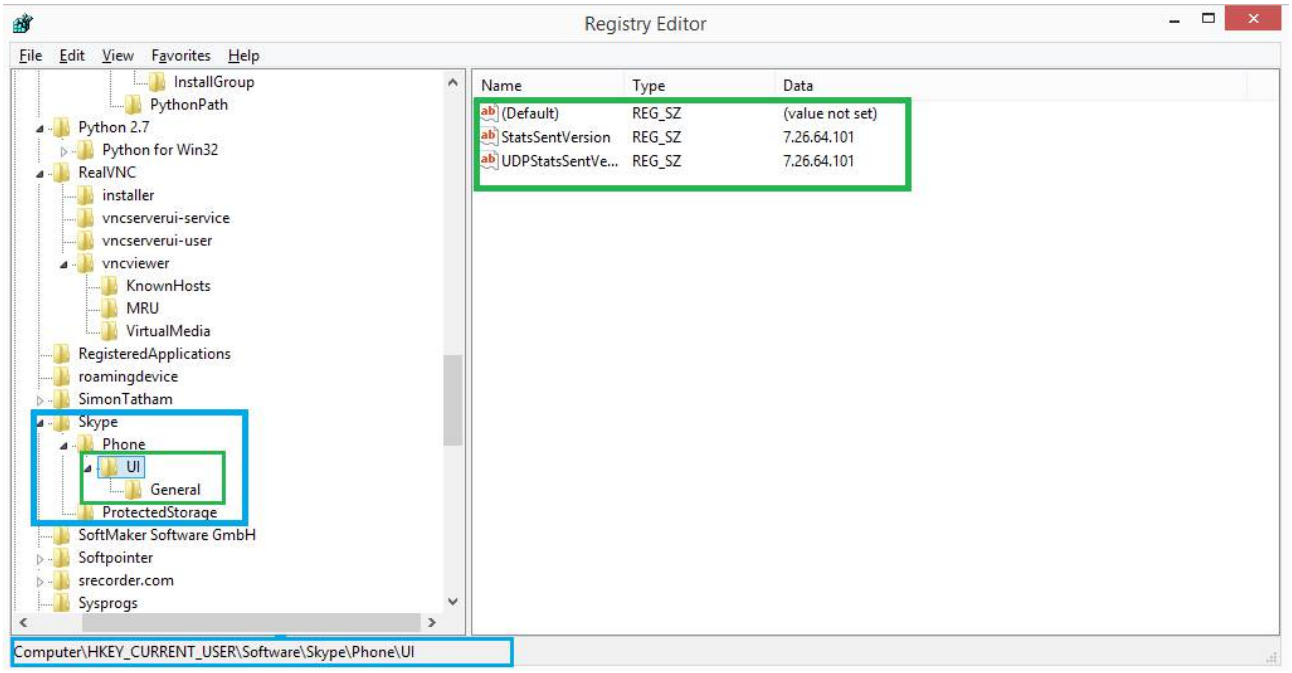
Remember, while accessing keys, you might see WindowsError: [Error 5] Access is denied exception, for some of the keys. Not some, in fact, for many of the keys. This is because, Windows Account Control (UAC), doesn’t give KEY_ALL_ACCESS on those keys, even for an Administrator. KEY_READ also throws the same exception, for few of the Keys as shown in the below example.

```
1 >>> import winreg
2 >>> a = winreg.OpenKey(winreg.HKEY_USERS, 'S-1-5-18')
3 >>> a = winreg.OpenKey(winreg.HKEY_USERS, 'S-1-5-19')
4 Traceback (most recent call last):
5 File "<string>" line 1, in <fragment>
6 WindowsError: [Error 5] Access is denied
7 >>> a = winreg.OpenKey(winreg.HKEY_USERS, 'S-1-5-20')
8 Traceback (most recent call last):
9 File "<string>";, line 1, in <fragment>
10WindowsError: [Error 5] Access is denied
```

As you can see, on the above code snippet, I can access S-1-5-18 easily, but, while trying to access S-1-5-19 or S-1-5-20, Access Denied error is thrown.

The next two methods that were of my use, are pretty much similar – EnumKey and EnumValue which are used to enumerate over the subkeys and values respectively for a given key. Both the methods accept the key and the index whose key or value you want to find. For a better understanding, let’s have a look at following image.

Here, if you look at the path in the bottom, HKEY_CURRENT_USER is the Registry constant and ‘Skype ‘ is a subkey in the given constant. When, We do Enumkey (with index=0) on Skype, it will give us it’s subkeys, i.e. Phone. Likewise, if we invoke EnumValue (with index=0) on Skype\Phone\UI, it will give us it’s first value [i.e. (‘StatsSentVersion’, ‘7.26.64.101’, 1)], in form of a tuple.



So both `Enumkey` and `EnumValue` throws WindowsError, in case of failure. Failures might be when there is no key or value existing for your provided index. To get all the subkeys for a given key, you have to increment the index one by one and invoke `Enumkey` until you encounter `WindowsError`. The same goes for `EnumValue` to find out all the values for a given key.

You can find out the same code on my Github. Here, it iterates over all available subkeys or values, increasing the counter by 1 until each keys or values are traversed.

The _winreg module

http://effbot.org/librarybook/winreg.htm

(New in 2.0) This module provides a low-level interface to the Windows registry database. The interface directly mirrors the underlying Windows API.

Example: Using the _winreg module

```
# File: winreg-example-1.py

import _winreg

explorer = _winreg.OpenKey(
    _winreg.HKEY_CURRENT_USER,
    "Software\\Microsoft\\Windows\\CurrentVersion\\Explorer"
    )

# list values owned by this registry key
try:
    i = 0
    while 1:
```

```
        name, value, type = _winreg.EnumValue(explorer, i)
        print repr(name),
        i += 1
except WindowsError:
    print

value, type = _winreg.QueryValueEx(explorer, "Logon User Name")

print
print "user is", repr(value)


$ python winreg-example-1.py
'Logon User Name' 'CleanShutdown' 'ShellState' 'Shutdown Setting'
'Reason Setting' 'FaultCount' 'FaultTime' 'IconUnderline' ...

user is u'Effbot'
```

Python *winreg* Module Examples

http://www.programcreek.com/python/index/2100/winreg

Subprocess Module

http://www.bogotobogo.com/python/python_subprocess_module.php

Site truffé de pubs, dommage...

SubProcess

https://docs.python.org/3/library/subprocess.html

17.5. `subprocess` — Subprocess management

Source code: Lib/subprocess.py

The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the `subprocess` module can be used to replace these modules and functions can be found in the following sections.

See also

PEP 324 – PEP proposing the subprocess module

17.5.1. Using the `subprocess` Module

The recommended approach to invoking subprocesses is to use the `run()` function for all use cases it can handle. For more advanced use cases, the underlying `Popen` interface can be used directly.

The `run()` function was added in Python 3.5; if you need to retain compatibility with older versions, see the Older high-level API section.

`subprocess.run`(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*)

> Run the command described by *args*. Wait for command to complete, then return a `CompletedProcess` instance.
>
> The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - apart from *timeout*, *input* and *check*, all the arguments to this function are passed through to that interface.
>
> This does not capture stdout or stderr by default. To do so, pass `PIPE` for the *stdout* and/or *stderr* arguments.
>
> The *timeout* argument is passed to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-

raised after the child process has terminated.

The *input* argument is passed to `Popen.communicate()` and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *universal_newlines* is true. When used, the internal `Popen` object is automatically created with `stdin=PIPE`, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a `CalledProcessError` exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *universal_newlines* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the `io.TextIOWrapper` default. Otherwise, file objects are opened in binary mode.

Examples:

```
>>>
```

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit
status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

New in version 3.5.

Changed in version 3.6: Added *encoding* and *errors* parameters

class `subprocess.CompletedProcess`

The return value from `run()`, representing a process that has finished.

`args`

The arguments used to launch the process. This may be a list or a string.

`returncode`

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

`stdout`

Captured stdout from the child process. A bytes sequence, or a string if `run()` was called with an encoding or errors. `None` if stdout was not captured.

If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and `stderr` will be `None`.

`stderr`

Captured stderr from the child process. A bytes sequence, or a string if `run()` was called with an encoding or errors. `None` if stderr was not captured.

`check_returncode()`

If `returncode` is non-zero, raise a `CalledProcessError`.

New in version 3.5.

`subprocess.DEVNULL`

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that the special file `os.devnull` will be used.

New in version 3.3.

`subprocess.PIPE`

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that a pipe to the standard stream should be opened. Most useful with `Popen.communicate()`.

`subprocess.STDOUT`

Special value that can be used as the *stderr* argument to `Popen` and indicates that standard error should go into the same handle as standard output.

exception `subprocess.SubprocessError`

Base class for all other exceptions from this module.

New in version 3.3.

exception `subprocess.TimeoutExpired`

Subclass of `SubprocessError`, raised when a timeout expires while waiting for a child process.

`cmd`

Command that was used to spawn the child process.

`timeout`

Timeout in seconds.

`output`

Output of the child process if it was captured by `run()` or `check_output()`. Otherwise, `None`.

`stdout`

Alias for output, for symmetry with `stderr`.

`stderr`

Stderr output of the child process if it was captured by `run()`. Otherwise, `None`.

New in version 3.3.

Changed in version 3.5: *stdout* and *stderr* attributes added

exception `subprocess.CalledProcessError`

Subclass of `SubprocessError`, raised when a process run by `check_call()` or `check_output()` returns a non-zero exit status.

`returncode`

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

`cmd`

Command that was used to spawn the child process.

`output`

Output of the child process if it was captured by `run()` or `check_output()`. Otherwise, `None`.

`stdout`

Alias for output, for symmetry with `stderr`.

`stderr`

Stderr output of the child process if it was captured by `run()`. Otherwise, `None`.

Changed in version 3.5: *stdout* and *stderr* attributes added

17.5.1.1. Frequently Used Arguments

To support a wide variety of use cases, the `Popen` constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be `True` (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

If *encoding* or *errors* are specified, or *universal_newlines* is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for `io.TextIOWrapper`.

For *stdin*, line ending characters `'\n'` in the input will be converted to the default line separator `os.linesep`. For *stdout* and *stderr*, all line endings in the output will be converted to `'\n'`. For more information see the documentation of the `io.TextIOWrapper` class when the *newline* argument to its constructor is `None`.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

New in version 3.6: Added *encoding* and *errors* parameters.

Note

The newlines attribute of the file objects `Popen.stdin`, `Popen.stdout` and `Popen.stderr` are not updated by the `Popen.communicate()` method.

If *shell* is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a

user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, glob, fnmatch, os.walk(), os.path.expandvars(), os.path.expanduser(), and shutil).

Changed in version 3.3: When *universal_newlines* is `True`, the class uses the encoding locale.getpreferredencoding(False) instead of `locale.getpreferredencoding()`. See the io.TextIOWrapper class for more information on this change.

Note

Read the Security Considerations section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the Popen constructor documentation.

17.5.1.2. Popen Constructor

The underlying process creation and management in this module is handled by the Popen class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class `subprocess.Popen`(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=False*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, *, *encoding=None*, *errors=None*)

Execute a child program in a new process. On POSIX, the class uses os.execvp()-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to Popen are as follows.

args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

Note

shlex.split() can be useful when determining the correct tokenization for *args*, especially in complex cases:

```
>>>
```

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
```

```
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd',
"echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying `CreateProcess()` operates on strings.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, `Popen` does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **dir** or **copy**). You do not need `shell=True` to run a batch file or console-based executable.

Note

Read the Security Considerations section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the `open()` function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

Changed in version 3.3.1: *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom

needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **ps**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

Warning

The *preexec_fn* parameter is not safe to use in the presence of threads in your application. The child process could deadlock before exec is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

Note

If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* parameter can take the place of a previously common use of *preexec_fn* to call os.setsid() in the child.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. (POSIX only). The default varies by platform: Always true on POSIX. On Windows it is true when *stdin*/*stdout*/*stderr* are `None`, false otherwise. On Windows, if *close_fds* is true then no handles will be inherited by the child process. Note that on Windows, you cannot set *close_fds* to true and also redirect the standard handles by setting *stdin*, *stdout* or *stderr*.

Changed in version 3.2: The default for *close_fds* was changed from `False` to what is described above.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be `True`. (POSIX only)

New in version 3.2: The *pass_fds* parameter was added.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a `str` and path-like object. In particular, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Changed in version 3.6: *cwd* parameter accepts a path-like object.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

Changed in version 3.2: *restore_signals* was added.

If *start_new_session* is true the setsid() system call will be made in the child process prior to the execution of the subprocess. (POSIX only)

Changed in version 3.2: *start_new_session* was added.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment.

Note

If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in Frequently Used Arguments. If *universal_newlines* is `True`, they are opened in text mode with default encoding. Otherwise, they are opened as binary streams.

New in version 3.6: *encoding* and *errors* were added.

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be `CREATE_NEW_CONSOLE` or `CREATE_NEW_PROCESS_GROUP`. (Windows only)

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Changed in version 3.2: Added context manager support.

Changed in version 3.6: Popen destructor now emits a `ResourceWarning` warning if the child process is still running.

17.5.1.3. Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called `child_traceback`, which is a string containing traceback information from the child's point of view.

The most common exception raised is `OSError`. This occurs, for example, when trying to execute

a non-existent file. Applications should prepare for OSError exceptions.

A ValueError will be raised if Popen is called with invalid arguments.

check_call() and check_output() will raise CalledProcessError if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as call() and Popen.communicate() will raise TimeoutExpired if the timeout expires before the process exits.

Exceptions defined in this module all inherit from SubprocessError.

New in version 3.3: The SubprocessError base class was added.

17.5.2. Security Considerations

Unlike some other popen functions, this implementation will never implicitly call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities.

When using `shell=True`, the shlex.quote() function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

17.5.3. Popen Objects

Instances of the Popen class have the following methods:

`Popen.poll()`

Check if child process has terminated. Set and return returncode attribute.

`Popen.wait(`*timeout*`=`*None*`)`

Wait for child process to terminate. Set and return returncode attribute.

If the process does not terminate after *timeout* seconds, raise a TimeoutExpired exception. It is safe to catch this exception and retry the wait.

Note

This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use Popen.communicate() when using pipes to avoid that.

Note

The function is implemented using a busy loop (non-blocking call and short sleeps). Use the asyncio module for an asynchronous wait: see asyncio.create_subprocess_exec.

Changed in version 3.3: *timeout* was added.

Deprecated since version 3.4: Do not use the *endtime* parameter. It is was unintentionally exposed in 3.3 but was left undocumented as it was intended to be private for internal use. Use *timeout* instead.

`Popen.communicate`(*input=None*, *timeout=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a `TimeoutExpired` exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

Note

The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: *timeout* was added.

`Popen.send_signal`(*signal*)

Sends the signal *signal* to the child.

Note

On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes *CREATE_NEW_PROCESS_GROUP*.

`Popen.terminate`()

Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the

Win32 API function `TerminateProcess()` is called to stop the child.

`Popen.kill()`

Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows `kill()` is an alias for `terminate()`.

The following attributes are also available:

`Popen.args`

The *args* argument as it was passed to `Popen` – a sequence of program arguments or else a single string.

New in version 3.3.

`Popen.stdin`

If the *stdin* argument was `PIPE`, this attribute is a writeable stream object as returned by `open()`. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not `PIPE`, this attribute is `None`.

`Popen.stdout`

If the *stdout* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not `PIPE`, this attribute is `None`.

`Popen.stderr`

If the *stderr* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not `PIPE`, this attribute is `None`.

Warning

Use `communicate()` rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

`Popen.pid`

The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

`Popen.returncode`

The child return code, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn't terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

17.5.4. Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

class `subprocess.STARTUPINFO`

Partial support of the Windows STARTUPINFO structure is used for `Popen` creation.

`dwFlags`

A bit field that determines whether certain `STARTUPINFO` attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES |
subprocess.STARTF_USESHOWWINDOW
```

`hStdInput`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard input handle for the process. If `STARTF_USESTDHANDLES` is not specified, the default for standard input is the keyboard buffer.

`hStdOutput`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

`hStdError`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

`wShowWindow`

If `dwFlags` specifies `STARTF_USESHOWWINDOW`, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

`SW_HIDE` is provided for this attribute. It is used when `Popen` is called with

shell=True.

17.5.4.1. Constants

The subprocess module exposes the following constants.

subprocess.STD_INPUT_HANDLE

The standard input device. Initially, this is the console input buffer, CONIN$.

subprocess.STD_OUTPUT_HANDLE

The standard output device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.STD_ERROR_HANDLE

The standard error device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.SW_HIDE

Hides the window. Another window will be activated.

subprocess.STARTF_USESTDHANDLES

Specifies that the STARTUPINFO.hStdInput, STARTUPINFO.hStdOutput, and STARTUPINFO.hStdError attributes contain additional information.

subprocess.STARTF_USESHOWWINDOW

Specifies that the STARTUPINFO.wShowWindow attribute contains additional information.

subprocess.CREATE_NEW_CONSOLE

The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.CREATE_NEW_PROCESS_GROUP

A Popen creationflags parameter to specify that a new process group will be created. This flag is necessary for using os.kill() on the subprocess.

This flag is ignored if CREATE_NEW_CONSOLE is specified.

17.5.5. Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use run() in many cases, but lots of existing code calls these functions.

subprocess.call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*,

timeout=None)

Run the command described by *args*. Wait for command to complete, then return the `returncode` attribute.

This is equivalent to:

```
run(...).returncode
```

(except that the *input* and *check* parameters are not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note

Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

`subprocess.check_call`(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

This is equivalent to:

```
run(..., check=True)
```

(except that the *input* parameter is not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note

Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

`subprocess.check_output`(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *encoding=None*, *errors=None*, *universal_newlines=False*, *timeout=None*)

Run command with arguments and return its output.

If the return code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute and any output in the `output` attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *universal_newlines* to `True` as described above in Frequently Used Arguments.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>>
```

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 3.1.

Changed in version 3.3: *timeout* was added.

Changed in version 3.4: Support for the *input* keyword argument was added.

17.5.6. Replacing Older Functions with the `subprocess` Module

In this section, "a becomes b" means that b can be used as a replacement for a.

Note

All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise `OSError` instead.

In addition, the replacements using `check_output()` will fail with a `CalledProcessError` if the requested operation produces a non-zero return code. The output is still available as the `output` attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from

the subprocess module.

17.5.6.1. Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

17.5.6.2. Replacing shell pipeline

```
output=`dmesg | grep hda`
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

17.5.6.3. Replacing os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

17.5.6.4. Replacing the os.spawn family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

17.5.6.5. Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)

(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)

(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

17.5.6.6. Replacing functions from the popen2 module

Note

If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)

(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` and `popen2.Popen4` basically work as `subprocess.Popen`, except that:

- `Popen` raises an exception if the execution fails.
- the *capturestderr* argument is replaced with the *stderr* argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with `Popen` to guarantee this behavior on all platforms or past Python versions.

17.5.7. Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.getstatusoutput`(*cmd*)

Return `(status, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(status, output)`. The locale encoding is used; see the notes on Frequently Used Arguments for more details.

A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function `wait()`. Example:

```
>>>
```

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
```

Availability: POSIX & Windows

Changed in version 3.3.4: Windows support added

`subprocess.getoutput`(*cmd*)

Return output (stdout and stderr) of executing *cmd* in a shell.

Like `getstatusoutput()`, except the exit status is ignored and the return value is a string containing the command's output. Example:

```
>>>
```

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: POSIX & Windows

Changed in version 3.3.4: Windows support added

17.5.8. Notes

17.5.8.1. Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See also

`shlex`
Module which provides function to parse and escape command lines.

Os.*

https://docs.python.org/3/library/os.html

os.exec*

`os.execl`(*path*, *arg0*, *arg1*, ...)
`os.execle`(*path*, *arg0*, *arg1*, ..., *env*)
`os.execlp`(*file*, *arg0*, *arg1*, ...)
`os.execlpe`(*file*, *arg0*, *arg1*, ..., *env*)
`os.execv`(*path*, *args*)
`os.execve`(*path*, *args*, *env*)
`os.execvp`(*file*, *args*)
`os.execvpe`(*file*, *args*, *env*)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as `OSError` exceptions.

The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or `os.fsync()` before calling an `exec*` function.

The "l" and "v" variants of the `exec*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `execl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `execl()`, `execle()`, `execv()`, and `execve()`, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For `execle()`, `execlpe()`, `execve()`, and `execvpe()` (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process' environment); the functions `execl()`, `execlp()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of the current process.

For `execve()` on some platforms, *path* may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

Availability: Unix, Windows.

New in version 3.3: Added support for specifying an open file descriptor for *path* for `execve()`.

Changed in version 3.6: Accepts a path-like object.

os.popen

`os.popen`(*cmd*, *mode='r'*, *buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument has the same meaning as the corresponding argument to the built-in `open()` function. The returned file object reads or writes text strings rather than bytes.

The `close` method returns `None` if the subprocess exited successfully, or the subprocess's return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be `- signal.SIGKILL` if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

This is implemented using `subprocess.Popen`; see that class's documentation for more powerful ways to manage and communicate with subprocesses.

os.spawn

`os.spawnl`(*mode*, *path*, *...*)
`os.spawnle`(*mode*, *path*, *...*, *env*)
`os.spawnlp`(*mode*, *file*, *...*)
`os.spawnlpe`(*mode*, *file*, *...*, *env*)
`os.spawnv`(*mode*, *path*, *args*)
`os.spawnve`(*mode*, *path*, *args*, *env*)
`os.spawnvp`(*mode*, *file*, *args*)
`os.spawnvpe`(*mode*, *file*, *args*, *env*)

Execute the program *path* in a new process.

(Note that the `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions.

Check especially the Replacing Older Functions with the subprocess Module section.)

If *mode* is `P_NOWAIT`, this function returns the process id of the new process; if *mode* is `P_WAIT`, returns the process's exit code if it exits normally, or `-signal`, where *signal* is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the `waitpid()` function.

The "l" and "v" variants of the `spawn*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (`spawnlp()`, `spawnlpe()`, `spawnvp()`, and `spawnvpe()`) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the `spawn*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `spawnl()`, `spawnle()`, `spawnv()`, and `spawnve()`, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For `spawnle()`, `spawnlpe()`, `spawnve()`, and `spawnvpe()` (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process' environment); the functions `spawnl()`, `spawnlp()`, `spawnv()`, and `spawnvp()` all cause the new process to inherit the environment of the current process. Note that keys and values in the *env* dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of `127`.

As an example, the following calls to `spawnlp()` and `spawnvpe()` are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. `spawnlp()`, `spawnlpe()`, `spawnvp()` and `spawnvpe()` are not available on Windows. `spawnle()` and `spawnve()` are not thread-safe on Windows; we advise you to use the `subprocess` module instead.

Changed in version 3.6: Accepts a path-like object.

os.system

`os.system`(*command*)

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `sys.stdin`, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`. Note that POSIX does not specify the meaning of the return value of the C `system()` function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable `COMSPEC`: it is usually **cmd.exe**, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the Replacing Older Functions with the subprocess Module section in the `subprocess` documentation for some helpful recipes.

Availability: Unix, Windows.