CHAPTER

THIRTYFIVE

MS WINDOWS SPECIFIC SERVICES

This chapter describes modules that are only available on MS Windows platforms.

35.1 `msilib` — Read and write Microsoft Installer files

Source code: Lib/msilib/__init__.py

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.FCICreate`(*cabname*, *files*)
: Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

  The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

  Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.UuidCreate`()
: Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.OpenDatabase`(*path*, *persist*)
: Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY`, or `MSIDBOPEN_TRANSACT`, and may include the flag `MSIDBOPEN_PATCHFILE`. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

`msilib.CreateRecord`(*count*)
: Return a new record object by calling `MSICreateRecord()`. *count* is the number of fields of the record.

`msilib.init_database`(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
: Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing `tables` and `_Validation_records` attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

`msilib.add_data(`*database*, *table*, *records*`)`
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. `'Feature'`, `'File'`, `'Component'`, `'Dialog'`, `'Control'`, etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, `None` can be passed.

Field values can be ints, strings, or instances of the Binary class.

`class msilib.Binary(`*filename*`)`
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

`msilib.add_tables(`*database*, *module*`)`
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

`msilib.add_stream(`*database*, *name*, *path*`)`
Add the file *path* into the `_Stream` table of *database*, with the stream name *name*.

`msilib.gen_uuid()`
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).

See also:

FCICreate UuidCreate UuidToString

35.1.1 Database Objects

`Database.OpenView(`*sql*`)`
Return a view object, by calling `MSIDatabaseOpenView()`. *sql* is the SQL statement to execute.

`Database.Commit()`
Commit the changes pending in the current transaction, by calling `MSIDatabaseCommit()`.

`Database.GetSummaryInformation(`*count*`)`
Return a new summary information object, by calling `MsiGetSummaryInformation()`. *count* is the maximum number of updated values.

`Database.Close()`
Close the database object, through `MsiCloseHandle()`.

New in version 3.7.

See also:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle

35.1.2 View Objects

`View.Execute(`*params*`)`
Execute the SQL query of the view, through `MSIViewExecute()`. If *params* is not `None`, it is a record describing actual values of the parameter tokens in the query.

View.GetColumnInfo(*kind*)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). *kind* can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.Fetch()
Return a result record of the query, through calling MsiViewFetch().

View.Modify(*kind*, *data*)
Modify the view, by calling MsiViewModify(). *kind* can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View.Close()
Close the view, through MsiViewClose().

See also:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

35.1.3 Summary Information Objects

SummaryInformation.GetProperty(*field*)
Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.GetPropertyCount()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.SetProperty(*field*, *value*)
Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in *GetProperty()*, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation.Persist()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

See also:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

35.1.4 Record Objects

Record.GetFieldCount()
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.GetInteger(*field*)
Return the value of *field* as an integer where possible. *field* must be an integer.

Record.GetString(*field*)
Return the value of *field* as a string where possible. *field* must be an integer.

Record.SetString(*field*, *value*)
Set *field* to *value* through MsiRecordSetString(). *field* must be an integer; *value* a string.

`Record.SetStream`(*field*, *value*)
Set *field* to the contents of the file named *value*, through `MsiRecordSetStream()`. *field* must be an integer; *value* a string.

`Record.SetInteger`(*field*, *value*)
Set *field* to *value* through `MsiRecordSetInteger()`. Both *field* and *value* must be an integer.

`Record.ClearData`()
Set all fields of the record to 0, through `MsiRecordClearData()`.

See also:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

35.1.5 Errors

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

35.1.6 CAB Objects

`class msilib.CAB`(*name*)
The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

`append`(*full*, *file*, *logical*)
Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

`commit`(*database*)
Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

35.1.7 Directory Objects

`class msilib.Directory`(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])
Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be `None`), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

`start_component`(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)
Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

`add_file`(*file*, *src=None*, *version=None*, *language=None*)
Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the

src file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*, *exclude=None*)
Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()
Remove `.pyc` files on uninstall.

See also:

Directory Table File Table Component Table FeatureComponents Table

35.1.8 Features

class msilib.Feature(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)
Add a new record to the `Feature` table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the `start_component()` method of *Directory*.

set_current()
Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

See also:

Feature Table

35.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class msilib.Control(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*, *condition=1*, *ordering=None*)
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Make an entry into the `ControlCondition` table for this control.

class msilib.RadioButtonGroup(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class msilib.Dialog(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

See also:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

35.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.schema
This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.sequence
This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.text
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

35.2 `msvcrt` — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the *getpass* module uses this in the implementation of the *getpass()* function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

Changed in version 3.3: Operations in this module now raise *OSError* where *IOError* was raised.

35.2.1 File Operations

msvcrt.locking(*fd*, *mode*, *nbytes*)
Lock part of a file based on file descriptor *fd* from the C runtime. Raises *OSError* on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

msvcrt.LK_LOCK
msvcrt.LK_RLCK
Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, *OSError* is raised.

msvcrt.LK_NBLCK
msvcrt.LK_NBRLCK
Locks the specified bytes. If the bytes cannot be locked, *OSError* is raised.

msvcrt.LK_UNLCK
Unlocks the specified bytes, which must have been previously locked.

msvcrt.setmode(*fd*, *flags*)
Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be *os.O_TEXT*; for binary, it should be *os.O_BINARY*.

msvcrt.open_osfhandle(*handle*, *flags*)
Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of *os.O_APPEND*, *os.O_RDONLY*, and *os.O_TEXT*. The returned file descriptor may be used as a parameter to *os.fdopen()* to create a file object.

msvcrt.get_osfhandle(*fd*)
Return the file handle for the file descriptor *fd*. Raises *OSError* if *fd* is not recognized.

35.2.2 Console I/O

msvcrt.kbhit()
Return true if a keypress is waiting to be read.

msvcrt.getch()
Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.getwch()
Wide char variant of *getch()*, returning a Unicode value.

msvcrt.getche()
Similar to *getch()*, but the keypress will be echoed if it represents a printable character.

msvcrt.getwche()
Wide char variant of *getche()*, returning a Unicode value.

msvcrt.putch(*char*)
Print the byte string *char* to the console without buffering.

msvcrt.putwch(*unicode_char*)
Wide char variant of *putch()*, accepting a Unicode value.

msvcrt.ungetch(*char*)
Cause the byte string *char* to be "pushed back" into the console buffer; it will be the next character read by *getch()* or *getche()*.

msvcrt.ungetwch(*unicode_char*)
Wide char variant of *ungetch()*, accepting a Unicode value.

35.2.3 Other Functions

msvcrt.heapmin()
Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises *OSError*.

35.3 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*.

35.3.1 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.

35.3.2 Constants

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

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.

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.

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.

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.

35.3.3 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.

35.4 winsound — Sound-playing interface for Windows

The *winsound* module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.Beep(*frequency*, *duration*)
Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, *RuntimeError* is raised.

winsound.PlaySound(*sound*, *flags*)
Call the underlying PlaySound() function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or None. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is None, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.MessageBeep(*type=MB_OK*)
Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, *RuntimeError* is raised.

winsound.SND_FILENAME
The *sound* parameter is the name of a WAV file. Do not use with *SND_ALIAS*.

winsound.SND_ALIAS
The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless *SND_NODEFAULT* is also specified. If no default sound is registered, raise *RuntimeError*. Do not use with *SND_FILENAME*.

All Win32 systems support at least the following; most systems support many more:

PlaySound() name	Corresponding Control Panel Sound name
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

For example:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

`winsound.SND_LOOP`
Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

`winsound.SND_MEMORY`
The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a *bytes-like object*.

Note: This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

`winsound.SND_PURGE`
Stop playing all instances of the specified sound.

Note: This flag is not supported on modern Windows platforms.

`winsound.SND_ASYNC`
Return immediately, allowing sounds to play asynchronously.

`winsound.SND_NODEFAULT`
If the specified sound cannot be found, do not play the system default sound.

`winsound.SND_NOSTOP`
Do not interrupt sounds currently playing.

`winsound.SND_NOWAIT`
Return immediately if the sound driver is busy.

Note: This flag is not supported on modern Windows platforms.

`winsound.MB_ICONASTERISK`
Play the `SystemDefault` sound.

`winsound.MB_ICONEXCLAMATION`
Play the `SystemExclamation` sound.

winsound.MB_ICONHAND
: Play the SystemHand sound.

winsound.MB_ICONQUESTION
: Play the SystemQuestion sound.

winsound.MB_OK
: Play the SystemDefault sound.