EDK II Python User's Manual

April, 2012
Revision 1.0

Acknowledgements

Revision History

Revision	Revision History	Date
0.1	Initial draft.	April 2012
0.8	First Review	April 2012
1.0	First Release	

Table of Contents

1 Preface

A Python* interpreter has been available for EFI environments since EFI 1.10 with Python 1.5.2. Since then, EFI has grown a standards body, The UEFI Forum, and an open source community on www.tianocore.org.

The first UEFI compliant version of the open source firmware was the EFI Development Kit, or EDK. (Sometimes referred to as EDK 1) An EDK version of Python 2.4 became available in 2006 as part of the EDK Toolkit 2.0 project.

This manual provides information about the Python 2.7 implementation for EDK II, the second major release of the EDK. This implementation is compliant with the UEFI Specification 2.3.1 with Errata A and is a native EDK II application.

At the time this manual was written, the supported version of Python was 2.7.2. Examples are written assuming version 2.7.2. It is assumed that these examples will be relevant for subsequent versions of Python 2.7.

1.1 Target Audience

Developers who want to build or use Python within a UEFI firmware environment should find this manual useful. It is assumed that the reader has familiarity with Python in a Microsoft* Windows or Linux* environment and is able to download and build the EDK II firmware.

Test or Validation engineers will find the EDK II Python scripting environment an asset for testing or provisioning of UEFI compliant target systems.

1.2 Document Organization

This manual does not attempt to document Python or EDK II. Instead, it focuses on presenting a practical guide for the EDK II specific aspects of building, installing, customizing, and using Python. A detailed overview of the EDK II specific modifications present in this port of Python is also provided.

Preface

Chapter 1, this chapter, describes the purpose, scope, and organization of the *EDK II Python User's Manual*.

Getting Started

Chapter 2 covers downloading and installing the Python 2.7 sources on the host system, configuring the set of built-in modules, and building Python as an UEFI application.

Installing Python

Chapter 3 shows how to install Python on the target system.

Target Customization

Chapter 4 identifies the site-specific package and script directories, path configuration files (name.pth), and importable zip archives that can be used to customize a target installation.

This chapter is a compilation of information provided in the Python 2.7 documentation on www.python.org, where it is distributed among the documentation for several modules.

The EDK II Implementation

Chapter 5 concludes this manual with a description of the EDK II specific changes and features of this implementation of Python.

1.3 Related Information

The following publications and sources of information may be useful to you or are referred to by this manual:

- *Unified Extensible Firmware Interface Specification*, Version 2.3.1 Errata A, Unified EFI, Inc, 2011, http://www.uefi.org.

The following publications are available at www.TianoCore.org:

- *EDK II Module Development Environment Package Library Specification*, Intel, 2009.
- *EDK II Platform Configuration Database Infrastructure Description, Version 0.55*, Intel, 2009.
- *EDK II C Coding Standards Specification*, Intel, 2011.
- *EDK II DEC File Specification*, Intel, 2011.
- *EDK II DSC File Specification*, Intel, 2011.
- *EDK II INF File Specification, Intel, 2011.*
- *EDK II Build Specification, Intel*, 2011.

Code documentation for the MdePkg, MdeModulePkg, IntelFrameworkPkg, and IntelFrameworkModulePkg packages, in CHM format, is available in the EDK II Documents repository at http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=Documents.

Python documentation and sources are available from http://www.python.org.

Python 2.7.3 was released on April 9, 2012, replacing 2.7.2. A copy of the appropriate portions of the Python 2.7.2 source distribution are now included in the *AppPkg/Applications/Python* directory of the EDK II source tree on edk2.tianocore.org.

1.4 Terms

The following terms are used throughout this document:

BNF

BNF is an acronym for "Backus Naur Form." John Backus and Peter Naur introduced, for the first time, a formal notation to describe the syntax of a given language.

Component

An executable image. Components defined in this specification support one of the defined module types.

DXE

Framework Driver Execution Environment phase.

DXE Runtime

Special class of DXE module that provides Runtime Services

EBNF

Extended “Backus-Naur Form” meta-syntax notation with the following additional constructs: square brackets “[…]” surround optional items, suffix “*” for a sequence of zero or more of an item, suffix “+” for one or more of an item, suffix “?” for zero or one of an item, curly braces “{…}” enclosing a list of alternatives and super/subscripts indicating between n and m occurrences.

EFI

Generic term that refers to one of the versions of the EFI specification: EFI 1.02, EFI 1.10, and UEFI 2.0 through the latest UEFI specification (2.3).

Extension Module

A module written in C or C++, using Python’s C API to interact with the core and with user code.

GUID

Globally Unique Identifier. A 128-bit value used to name entities uniquely. A unique GUID can be generated by an individual without the help of a centralized authority. This allows the generation of names that will never conflict, even among multiple, unrelated parties. GUID values can be registry format (8-4-4-4-12) or C data structure format.

Library Class

A library class defines the API or interface set for a library. The consumer of the library is coded to the library class definition. Library classes are defined via a library class .h file that is published by a package.

Library Instance

A specific implementation (instance) of one or more library classes.

Module (UEFI)

A module is either an executable image or a library instance. For a list of module types supported by this package, see module type.

Module (Python)

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

Module Type (UEFI)

All libraries and components belong to one of the following module types: BASE, SEC, PEI_CORE, PEIM, DXE_CORE, DXE_DRIVER, DXE_RUNTIME_DRIVER, DXE_SMM_DRIVER, DXE_SAL_DRIVER, UEFI_DRIVER, or UEFI_APPLICATION. These definitions provide a framework that is consistent with a similar set of requirements. A module that is of module type BASE, depends only on headers and libraries provided in the MDE, while a module that is of module type DXE_DRIVER depends on common DXE components. For a definition of the various module types, see module type.

Package (UEFI)

A package is a container. It can hold a collection of files for any given set of modules.

Packages may contain Source Modules, containing all source files and descriptions of a module; or Binary Modules, containing PI FFS Sections and a description file specific to linking and binary editing of features and attributes or both Binary and Source modules.

Multiple modules can also be combined into a package. Multiple Packages can also be Bundled into a single Distribution Package.

Package (Python)

A module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file __init__.py.

Protocol

An API named by a GUID as defined by the UEFI specification.

PCD

Platform Configuration Database.

PCD C Name

The symbolic name for a PCD Token that follows the ANSI C naming conventions for the name of a variable.

PCD Element

A single configurable element within the Platform Configuration Database, uniquely identified by a Token Space GUID and Token Number.

PCD Token Space GUID

The GUID value associated with a group of PCD Tokens. Using a GUID allows vendors to allocate their own Token Numbers for configuration elements that apply to their own modules, libraries or platforms without a centralized allocator. Within the Distribution Description file, a PCD Token Space GUID is referred to using the PCD Token Space GUID C Name.

PCD Token Space GUID C Name

A symbolic name for a PCD Token Space GUID value that follows the ANSI C naming conventions for the name of a variable.

Platform Configuration Database (PCD)

The collection of PCD elements that can be configured when building modules, libraries, or platform firmware images. These elements are identified by a Token Space GUID and Token Number. PCD elements are declared in packages by Package Developers. Modules Developers use PCD elements in the design of their modules to increase the portability of their modules to a wider array of platform targets. Platform Integrators set the values of PCD elements based on specific platform requirements. A Platform Integrator has many options when configuring PCDs for a specific platform. They may configure PCD elements to be set to static values at build time. They may also configure PCD elements so the binary image of a Module may be patched prior to integration into platform firmware images. They may also configure PCD elements so the binary image of platform firmware may be patched. They may also configure PCD elements so they can be accessed at runtime through the PCD services described in the PI 1.2 Specification.

Runtime Services

Interfaces that provide access to underlying platform-specific hardware that might be useful during OS runtime, such as time and date services. These services become active during the boot process but also persist after the OS loader terminates boot services.

SKU

Stock Keeping Unit.

UEFI Application

An application that follows the UEFI specification. The only difference between a UEFI application and a UEFI driver is that an application is unloaded from memory when it exits regardless of return status, while a driver that returns a successful return status is not unloaded when its entry point exits.

UEFI

Unified Extensible Firmware Interface

UEFI Specification

The UEFI Specification describes an interface between the operating system (OS) and the platform firmware. UEFI was preceded by the Extensible Firmware Interface Specification 1.10 (EFI). This specification is released by the Unified EFI Forum.

This document references the UEFI Specification 2.3.1, with errata A.

Unified EFI Forum

A non-profit collaborative trade organization formed to promote and manage the UEFI standard. For more information, see www.uefi.org.

1.5 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.5.1 Procedure descriptions

The procedures described in this document generally have the following format:

ProcedureName()

The formal name of the procedure.

Summary

A brief description of the procedure.

Prototype

A “C-style” procedure header defining the calling sequence.

Parameters

A brief description of each field in the procedure prototype.

Description

A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions

The type declarations and constants that are used only by this procedure.

Status Codes Returned

A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.5.2 Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be FIFO.

Pseudo code is presented in a C-like format, using C conventions where appropriate.

The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the UEFI Specification.

1.5.3 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	Any plain text that is underlined and in blue indicates an active link to a cross-reference. Click on the word to follow the hyperlink.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	Words in a **`Bold Monospace`** typeface, that is underlined and in blue, indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
$(VAR)	The symbol VAR defined by the utility or input files.

The following typographic conventions are used in this document to illustrate the Extended Backus-Naur Form.

[item]	Square brackets denote the enclosed item is optional.
{item}	Curly braces denote a choice or selection item, only one of which may occur on a given line.
<item>	Angle brackets denote a name for an item.
(range-range)	Parenthesis with characters and dash characters denote ranges of values, for example, (a-zA-Z0-9) indicates a single alphanumeric character, while (0-9) indicates a single digit.
"item"	Characters within quotation marks are the exact content of an item, as they must appear in the output text file.
?	The question mark denotes zero or one occurrences of an item.
*	The star character denotes zero or more occurrences of an item.
+	The plus character denotes one or more occurrences of an item.
item$^{\{n\}}$	A superscript number, n, is the number of occurrences of the item that must be used. Example: (0-9)^{8} indicates that there must be exactly eight digits, so 01234567 is valid, while 1234567 is not valid.
item$^{\{n,\}}$	A superscript number n, followed by a comma "," and within curly braces, indicates the minimum number of occurrences of item, with no maximum number of occurrences.
item$^{\{,n\}}$	A superscript number, n, preceded by a comma "," and within curly braces, indicates a maximum number of occurrences of item.

item$^{\{n,m\}}$	A super script number, n, followed by a comma ",“ and a number, m, with both enclosed within curly braces, indicates that the number of occurrences can be from n to m occurrences of item, inclusive.

1.6 Getting Help

Even with this manual, and the EDK II, UEFI, and Python documentation, you will eventually run into a question whose answer you can't figure out on your own. When that happens, there are several resources you can turn to for help.

The authoritative source for Python resources is www.python.org. These resources include mailing lists and documentation.

The gateway to the EDK II community is www.tianocore.org, or edk2.tianocore.org. There, you will be able to find documentation and source code for EDK II and related projects. For the EDK II port of Python and other EDK II related questions, turn to the edk2-devel@lists.sourceforge.net mail list.

Finally, you can find information about the UEFI body of standards at www.uefi.org.

2
Getting Started

2.1 Getting Python

These instructions cover how to get a working copy of the CPython source code (CPython is the version of Python available from http://www.python.org/). It will also give you an overview of relevant portions of the directory structure of the CPython source code.

2.1.1 Use the Source, Luke

Download the source code from http://www.python.org/download/releases/. Select the Python 2.7.x release to get the latest Python 2.7 distribution.

Currently only version 2.7.2 of the CPython distribution is supported. Since this in no longer available on the www.python.org web site, it has been included in the *AppPkg/Applications/Python/Python-2.7.2* subdirectory of the EDK II source tree.

If you downloaded a source distribution file, extract the file into the *AppPkg/Applications/Python* directory of your EDK II source tree. This should produce a directory, such as, *AppPkg/Applications/Python/Python-2.7.2*.

2.1.2 Installing Modified Files

Modified versions of files from supported CPython distributions are in the *Python/PyMod-#.#.#* directory, where the #.#.# is replaced with the version number of a supported distribution; currently 2.7.2. The PyMod sub-directories mirror the directory structure of the corresponding CPython distribution. Copy files from the PyMod sub-directories into the corresponding directories of the CPython distribution. This will overwrite existing files with files that have been modified to build with EDK II.

This step may be skipped if building CPython 2.7.2 since the modified files have already been copied into the Python-2.7.2 directory tree for you.

2.2 Building Python

One final step is needed before building Python for the first time: enabling Python in the build description file. To do this, edit *AppPkg/AppPkg.dsc* and enable (un-comment) the `PythonCore.inf` line within the `[Components]` section of the file.

That's it! You are now ready to build Python. Depending upon how you have configured your EDK II *Config/target.txt* file, one of the following style of build command will get the job done.

- build
- build -a X64
- build -p AppPkg\AppPkg.dsc -a IA32

2.3 Configuring Python

For convenience, this section will refer to files and directories relative to the Python application directory at `AppPkg/Applications/Python`.

2.3.1 Built-in Modules

By default, EDK II Python is configured with just the built-in modules needed to run the interpreter and *`pydoc`* for the help facility. Options exist to add built-in modules or even to strip Python down to the bare minimum. The configuration described here affects not only the capabilities of the resulting Python, but the size of the executable.

The file *`config.c`*, in the Efi directory, specifies which extension modules are built into the resulting Python.efi executable. Documentation on the content, use, and format of this file is available on www.python.org. The EDK II specific version will be described here.

The *`config.c`* file contains a number of external function declarations and one array definition: `_PyImport_Inittab`. Each element of this array is a structure consisting of an extension module name and a pointer to that module's initialization function. Python will only be able to use an extension module if it is described in this array.

The EDK II version of *`config.c`* has the entries in the `_PyImport_Inittab` array divided into three groups: optional modules, pydoc dependencies, and mandatory modules.

Optional modules are the first modules listed in the array. By default they are all disabled, or commented out. By enabling, or un-commenting, these entries; the associated extension module will be incorporated when Python is next built.

Pydoc dependency modules are the modules required by the `pydoc` help facility announced when Python first starts in interactive mode. See Section 2.3.3, below, for more information about these modules.

Mandatory modules consist of the modules between "`edk2`" and "`_warnings`", inclusive. These modules must be present to achieve a minimally useful Python. Unless you are an experienced Python hacker, these modules must always be enabled.

The only EDK II specific extension module is the "`edk2`" module. This module provides access to UEFI firmware functionality in a manner that is standardized by the C Standard and the POSIX standard. **This module should not be imported directly**. Instead, import the module `os`, which provides a *portable* version of this interface. On UEFI-based platforms, this module provides a subset of the POSIX functionality.

Detailed information on EDK II and UEFI specific features can be found in Chapter 5, The EDK II Implementation.

2.3.2 Feature Set

A more advanced capability exists for tuning Python's feature set by editing the **`pyconfig.h`** file. For EDK II, this file resides in the directory appropriate for the target processor's architecture: Ia32, Ipf, X64, etc. Normally, this file will not need to be modified.

A description of the contents of this file, and modification guidelines, are available on www.python.org.

2.3.3 Minimalist Python

If interactive help is not needed, you may eliminate the requirement for the pydoc dependent modules. This requires editing the *Lib/site.py* file, within the Python source tree, as well as disabling unneeded modules (those between "_codecs" and "time", inclusive) in **config.c**.

By commenting-out the call to sethelper(), in the main() function at line 508, the pydoc help facility is disabled. No other changes to *site.py* are required. For completeness, you may optionally edit the COPYRIGHT macro, defined in *Modules/main.c*, to eliminate the "help" portion of the message.

3 Installing Python

This chapter describes installation of Python on a target UEFI system. The required directory structure is described as well as installation options you may wish to consider.

3.1 Target Directory Structure

Python depends upon the existence of several directories and files on the target system.

```
\Efi                            Root of the UEFI system area.
 |- \Tools                      Location of the Python.efi executable.
 |- \Boot                       UEFI specified Boot directory.
 |- \StdLib                     Root of the Standard Libraries sub-tree.
    |- \etc                     Configuration files used by libraries.
    |- \tmp                     Temporary files created by tmpfile(), etc.
    |- \site-python             Site-specific packages and modules.
    |- \lib                     Root of the libraries tree.
       |- \python.27            Directory containing the Python library modules.
          |- \lib-dynload       Dynamically loadable Python extensions.
          |- \site-packages     Site-specific packages and modules.
```

The directories up to, and including, *\Efi\StdLib\tmp*, are required by the Standard C Library. The remaining directories are introduced in support of Python.

3.2 Target Population

Once these directories are created on the target system they are ready to be populated from the development system as follows.

- Copy **Python.efi** to *\Efi\Tools*.
- Copy desired Python library files from the *Python-2.7.x/Lib* directory to *\Efi\StdLib\lib\python.27*. The recommended minimum set of modules (.py, .pyc, and/or .pyo) are:

os	**stat**	**ntpath**	**warnings**	**traceback**
site	**types**	**copy_reg**	**linecache**	**genericpath**

When possible, it is recommended that only .pyc or .pyo files be used. This will result in the greatest space savings and faster execution. The Python library file **compileall.py** can be used to pre-compile Python modules prior to loading them onto the target system.

3.3 Python Library Modules

This is a partial list of the packages and modules of the Python Standard Library that have been tested or used in some manner.

Table 1: Python Library Modules

encodings	genericpath.py	sha.py
importlib	getopt.py	SimpleHTTPServer.py
json	hashlib.py	site.py
pydoc_data	heapq.py	socket.py
xml	HTMLParser.py	SocketServer.py
abc.py	inspect.py	sre.py
argparse.py	io.py	sre_compile.py
ast.py	keyword.py	sre_constants.py
atexit.py	linecache.py	sre_parse.py
BaseHTTPServer.py	locale.py	stat.py
binhex.py	md5.py	string.py
bisect.py	modulefinder.py	StringIO.py
calendar.py	ntpath.py	struct.py
cmd.py	numbers.py	textwrap.py
codecs.py	optparse.py	token.py
collections.py	os.py	tokenize.py
copy.py	platform.py	traceback.py
copy_reg.py	posixpath.py	types.py
csv.py	pydoc.py	warnings.py
dummy_thread.py	random.py	weakref.py
fileinput.py	re.py	xmllib.py
formatter.py	repr.py	zipfile.py
functools.py	runpy.py	

4 Target Customization

The Python library file, *site.py*, is responsible for setting up and processing several site-specific customization mechanisms. This chapter describes EDK II specific behavior.

4.1 Customization Directories

Up to four directories are added to the module search path, depending upon whether they exist or not. These directories consist of: *\Efi\StdLib\lib\python.27*, *\Efi\StdLib\lib\python.27\lib-dynload*, *\Efi\StdLib\lib\python.27\site-packages*, *\Efi\StdLib\lib\site-python*. As each of these directories is added, it is checked for path configuration files.

A path configuration file is a file whose name has the form *name.pth* and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is added to `sys.path` more than once. Blank lines and lines beginning with # are skipped. Lines starting with **`import`** (followed by space or tab) are executed.

4.2 Customization Files

After the above path manipulations are complete, an attempt is made to import a module named **`sitecustomize`**, which can perform arbitrary site-specific customizations. It is typically created by an administrator in the `site-packages` directory. If this import fails with an **`ImportError`** exception, it is silently ignored.

The EDK II implementation of Python does not support the **`usercustomize`** module, and the value of **`site.USER_SITE`** will be *None*.

4.3 Zip Archives

If the **`zlib`** and **`zipimport`** extension modules have been built in, see section 2.3.1, Built-in Modules, then Python will have the ability to import Python modules (*.py, *.py[co]) and packages from ZIP-format archives. With these two extension modules present, the standard **`import`** mechanism will search `sys.path` items that are paths to ZIP archives.

The *name.pth* files, as described in Section 4.1 above, are a convenient means of adding ZIP-format archives to `sys.path`.

Note: *It is important to note that* `site.{py, py[co]}` *and* `os.{py, py[co]}` *must not be imported from a ZIP-format archive and must exist in the* `\Efi\StdLib\lib\python.27` *directory.*

5 The EDK II Implementation

The EDK II implementation of CPython is written as a UEFI compliant application using libraries provided by the EDK II firmware. The Standard Library implementation in EDK II is used to provide the Standard C and POSIX APIs assumed by CPython. This resulted in very few behavioral differences and few changes to the CPython source code being needed.

5.1 CPython Code Changes

The changes that were made to the CPython sources fall into one of three categories:

1. EDK II specific file created. Changes to the file were so extensive that a new file was created specifically for EDK II.
2. Add “uefi” as a new “OS” type. A few files determine their behavior based upon the type of “OS” that Python is to be run under. Since none of these selections provided the proper set of functionality required, it was necessary to add “uefi” as a new “OS” type along with the appropriate behaviors.
3. Fix issues affecting portability between CPU architectures. Usually, this consisted of just adding an appropriate cast. It is intended that these changes will be submitted to the Python development teams for consideration for inclusion in a future CPython 2.7.x release.

The following sub-sections detail changes to specific key files that fit into the first two categories.

5.1.1 edk2module.c

This is a category-1 file. Starting with **`posixmodule.c`** as a base, functionality has been either removed, disabled, or enabled based upon services provided by the EDK II Standard C and POSIX libraries. Where code was conditional upon the host OS, it was replaced with either the existing generic code or code appropriate for EDK II.

The only functions with significant changes are `_pystat_fromstructstat()` and `posix_listdir()`. In both cases the changes were to accommodate differences between POSIX and the information available from the UEFI file system.

5.1.2 getpath.c

Possibly the most heavily modified file, **`getpath.c`** clearly fits into category 1. The purpose of the file is primarily to determine the module search path and the full path to the currently running Python executable.

It is assumed that directories in the module search path reside on the same UEFI volume as the Python executable.

This file is a candidate for significant future simplification.

5.1.3 ntpath.py

Because UEFI paths most closely resemble Microsoft Windows* (NT) paths, this implementation uses **`ntpath.py`**. Two changes were needed to **`ntpath.py`** in order to support "drive letters" longer than one character: `splitdrive()` now supports multiple character "drive letters" to the left of the ':' and `splitunc()` now properly handles paths with multiple character "drive letters".

Because it is debatable whether these changes fix a "bug" or not, it is currently considered a category-2 change.

5.1.4 site.py

The **`site`** module is automatically imported during Python initialization. This automatic import can be suppressed using the interpreter's `-S` option. Importing this module will append site-specific paths to the module search path and add a few builtins.

For this implementation, the following changes were made:

1. remove code not needed for this implementation
2. disable the per-user **`site-packages`** directory
3. change form of Python lib directory name from "python2.7" to "python.27"
4. use **`os.path.pathsep`** instead of hard coded ';' for the path separator

Otherwise, the overall logic, structure, and flow of the original file is retained. Since a completely new file was created, these are considered category-1 changes.

5.1.5 os.py

The only code change made to this file was to add one block to the cascading set of supported "operating systems". This block of code:

- imports the "edk2" extension module
- sets **`os.name`** and **`os.linesep`**
- sets **`os.path`**

The change made to this file defines category-2.

5.2 Execution Environment

When using Python within the UEFI environment, the following conditions should be noted.

1. Python may only be invoked from one of the EFI shells.
2. The current directory must reside on the same file-system volume as the Python Library or Python executable.
3. No input line editing.
4. Both '/' and '\' are accepted as directory separators in paths.
5. The path separator character is ';'
6. The environment is read-only with the old EDK Shell but read/write with the new UEFI Shell.
7. Changes to the environment do no persist between invocations of Python.