IBM Open Enterprise Python for z/OS, V3.8

User's Guide

IBM

SC28-3143-00 (2020-06-26 updated)

This edition applies to version 3.8 of IBM® Open Enterprise Python for z/OS® (order number: SC28-3143-00) and to all subsequent releases and modifications until otherwise indicated in new editions.

It is our intention to update the product documentation for this release periodically, without updating the order number. If you need to uniquely refer to the version of your product documentation, refer to the order number with the date of update.

Last updated: 2020-6-26

Contents

Chapter 1. Overview

IBM Open Enterprise Python for z/OS is an industry-standard Python interpreter for the z/OS platform. IBM Open Enterprise Python for z/OS leverages the latest z/Architecture® instructions to provide an exceptional implementation on the z/OS platform.

Python is one of the most popular and fastest growing programming languages. Simple programming syntax, a rich ecosystem of modules, the capability to interact with other languages and platforms, and strong community support across multiple industries drive its popularity.

IBM Open Enterprise Python for z/OS, V3.8 includes:

- The Python Standard Library, which provides an extensive set of functions that can save development resources in creating applications.
- Access to a growing collection of several thousand additional packages, available from the Python Package Index (PyPI).
- Support for ASCII, ECDIC, and Unicode character sets to provide users with choice of encodings.

Chapter 2. Installation and configuration

IBM Open Enterprise Python for z/OS is available in two installation formats, SMP/E and PAX. Select the installation format that applies to you:

- "Installing and configuring the SMP/E edition" on page 3
- "Installing and configuring the PAX edition" on page 4

For customization and environment configuration information, see "Customization and environment configuration" on page 5.

Installing and configuring the SMP/E edition

The Program Directory for the product details specific installation requirements and instructions in Chapter 5 and Chapter 6. For information about the latest APAR fixes, see Fix list for IBM Open Enterprise Python for z/OS.

The following checklist summarizes the key configuration steps for a successful installation.

Hardware requirements

- z15™
- z14®/z14 model ZR1
- z13®/z13s®
- zEnterprise® EC12/BC12

Software requirements

- z/OS UNIX System Services enabled on any of following operating systems:
  - z/OS V2R3
  - z/OS V2R4, or later
- Integrated Cryptographic Services Facility (ICSF) must be enabled on systems where IBM Open Enterprise Python for z/OS runs. For more information, see ICSF System Programmer's Guide (SC14-7507) and ICSF Administrator's Guide (SC14-7506).
- Optional: Packages that are installed with `pip` might include source that is written in other programming languages. It is your responsibility to ensure that compilers are available for other languages. Python looks for `/bin/xlc` in UNIX System Services to compile C modules, and for both `/bin/xlc` and `/bin/xlc++` to compile C++ modules. The Python interpreter by default uses `/bin/xlc` to link these modules.

Configuration

IBM Open Enterprise Python for z/OS is an OMVS-based application, which requires certain configuration on the z/OS UNIX System Services file system to ensure proper operation.

- Validate that `/usr/bin/env` exists. If not configured, refer to the instructions in "Customization and environment configuration" on page 5.
- Ensure that `/tmp` has at least 660 MB or more of disk space configured. To use an alternative file system, you can set the `TMPDIR` environment variable to a directory that has sufficient space.

Default installation location for IBM Open Enterprise Python for z/OS

The default Python SMP/E installation location on z/OS is `/usr/lpp/IBM/cyp/v3r8/pyz`.

Environment variables for SMP/E installation

Set the following environment variables before using IBM Open Enterprise Python for z/OS.

Configure the PATH and LIBPATH environment variables to include the bin directories for IBM Open Enterprise Python for z/OS with the following commands:

```
export PATH=/usr/lpp/IBM/cyp/v3r8/pyz/bin:$PATH
export LIBPATH=/usr/lpp/IBM/cyp/v3r8/pyz/lib:$LIBPATH
```

Set the auto conversion environment variables:

```
export _BPXK_AUTOCVT='ON'
export _CEE_RUNOPTS='FILETAG(AUTOCVT,AUTOTAG) POSIX(ON)'
```

Set the file tagging environment variables:

```
export _TAG_REDIR_ERR=txt
export _TAG_REDIR_IN=txt
export _TAG_REDIR_OUT=txt
```

Installing and configuring the PAX edition

The requirements for installing IBM Open Enterprise Python for z/OS, v3.8 are listed below.

Hardware requirements

- z15™
- z14®/z14 model ZR1
- z13/z13s
- zEnterprise EC12/BC12

Software requirements

- z/OS UNIX System Services enabled on any of following operating system:
  - z/OS V2R3
  - z/OS V2R4, or later
- You must enable the Integrated Cryptographic Services Facility (ICSF) on systems where IBM Open Enterprise Python for z/OS runs. For more information, see ICSF System Programmer's Guide (SC14-7507) and ICSF Administrator's Guide (SC14-7506).
- Optional: Packages that are installed with `pip` might include source that is written in other programming languages. It is your responsibility to ensure that compilers are available for other languages. Python looks for `/bin/xlc` in UNIX System Services to compile C modules, and for both `/bin/xlc` and `/bin/xlc++` to compile C++ modules. The Python interpreter by default uses `/bin/xlc` to link these modules.

Configuration

IBM Open Enterprise Python for z/OS is an OMVS-based application, which requires certain configuration on the z/OS UNIX System Services file system to ensure proper operation.

- Validate that `/usr/bin/env` exists. If not configured, refer to the instructions in "Customization and environment configuration" on page 5.
- Ensure that `/tmp` has at least 660 MB or more of disk space configured. To use an alternative file system, you can set the TMPDIR environment variable to a directory that has sufficient space.

Install the PAX archive file

- 250 MB is required to download the PAX archive file.

- Minimum 660 MB is required to extract and install Python.
- Create a directory `<mydir>` to hold the extracted PAX files.
- Unpax the downloaded file with the following command:

```
$ cd <mydir>
$ pax -p p -r -f <path to downloaded paxfile>
```

Environment variables for PAX archive installation

Set the following environment variables before using IBM Open Enterprise Python for z/OS.

Configure the PATH and `LIBPATH` environment variables to include the `bin` directories for IBM Open Enterprise Python for z/OS with the following commands:

```
export PATH=<path to install dir>/bin:$PATH
export LIBPATH=<path to install dir>/lib:$LIBPATH
```

Set the auto conversion environment variables:

```
export _BPXK_AUTOCVT='ON'
export _CEE_RUNOPTS='FILETAG(AUTOCVT,AUTOTAG) POSIX(ON)'
```

Set the file tagging environment variables:

```
export _TAG_REDIR_ERR=txt
export _TAG_REDIR_IN=txt
export _TAG_REDIR_OUT=txt
```

Customization and environment configuration

IBM Open Enterprise Python for z/OS requires `/usr/bin/env`, but your system might only have `/bin/env`. You can take the following steps to verify the path for the env command.

1. Ensure that `/usr/bin/env` exists and provides a correct listing of the environment. In an SSH or Telnet shell environment, run the following command to verify the location and contents of `env`. The command returns a list of name and value pairs for the environment in your shell.

   ```
   /usr/bin/env
   ```

   If `/usr/bin/env` does not exist, complete the following steps to set it up:

   a. Locate the env program on your system. A potential location is `/bin/env`.

   b. Create a symbolic link (symlink) so that `/usr/bin/env` resolves to the true location of env. For example:

      ```
      ln -s /bin/env /usr/bin/env
      ```

   c. In an SSH or Telnet shell environment, run the following command to verify if the symlink works. The command returns a list of name and value pairs for the environment in your shell.

      ```
      /usr/bin/env
      ```

2. Verify that the symbolic link for the env command persists across system IPLs.

   Depending on how `/usr/bin/` is configured on your system, the symbolic link for `/usr/bin/env` might not persist across an IPL without extra setup. Ensure that your IPL setup includes creation of this symbolic link, if necessary.

Optional: Set symlinks for `/usr/bin`. When using `pip`, some packages expect Python to be installed into `/usr/bin`. You can set symlinks by running the following commands:

1. ```
   ln -sf <install directory>/bin/python3 /usr/bin/python
   ```
   ```

2.
```
ln -sf <install directory>/bin/python3 /usr/bin/python3
```

3.
```
ln -sf <install directory>/bin/python3 /usr/bin/python3.8
```

Note that `<install directory>` in the above examples is the path you chose for installation.

Chapter 3. Getting started with IBM Open Enterprise Python for z/OS

Ensure the required environment variables are set before getting started with IBM Open Enterprise Python for z/OS. See “Environment variables for SMP/E installation” on page 4 or “Environment variables for PAX archive installation” on page 5.

Verify your Python version

Check your Python version with the following line:

```
$ python3 --version
```

Check your Python installation location with the following line:

```
$ python3 -c "import sys; print(sys.executable)"
```

"Hello world!" script

If the version number and executable path are correct, you are now ready to write your Python script.

For an EBCDIC (code page 1047) encoded file, perform the following steps:

```
$ vi test_script_ebcdic_enc.py


def main():
    print("hello world!")


if __name__ == "__main__":
    main()


$ chtag -tc IBM-1047 test_script_ebcdic_enc.py
$ python3 test_script_ebcdic_enc.py
```

For a UTF-8 encoded file, perform the following steps:

```
$ vi test_script_utf8_enc.py


def main():
    print("hello world!")


if __name__ == "__main__":
    main()


$ chtag -tc ISO8859-1 test_script_utf8_enc.py
$ python3 test_script_utf8_enc.py
```

The message is printed as follows:

```
hello world!
```

Note: Ensure that your scripts are tagged correctly to avoid syntax and encoding errors. For more information, see Chapter 6, “Codesets and translation,” on page 15.

Chapter 4. Package documentation for zos_util

The `zos_util` is an extended standard OS module Python package that allows users to set, reset, and display extended file attributes on z/OS.

Functions

zos_util.chtag(*filepath*, *ccsid=819*, *set_txtflag=True*)
: changes information in a file tag. A file tag is composed of a numeric coded character set identifier (*ccsid*) and a text flag (*set_txtflag*) codeset.

  set_txtflag = True indicates that the file has uniformly encoded text data.

  set_txtflag = False indicates that the file has non-uniformly encoded text data.

zos_util.untag(*filepath*)
: removes any tagging information that is associated with the file and sets the status of the file to untagged.

zos_util.get_tag_info(*filepath*)
: returns a tuple of file tagging information (ccsid, set_txtflag) associated with the file.

zos_util.tag_binary(*filepath*)
: changes the file tag to binary mode to indicate that the file contains only binary (non-uniformly encoded) data.

zos_util.tag_text(*filepath*)
: changes the file tag to text mode, which indicates that the specified file contains pure text (uniformly encoded) data.

  The existing *ccsid* that is associated with the file is retained.

zos_util.tag_mixed(*filepath*)
: changes the file tag to mixed mode, which indicates that the file contains mixed text and binary data.

  The existing *ccsid* that is associated with the file is retained.

zos_util.enable_apf(*filepath*)
: sets APF-authorized attribute on an executable program file (load module). It behaves as if the file is loaded from an APF-authorized library and raises `PermissionError` exception when the operation fails.

zos_util.disable_apf(*filepath*)
: unsets APF-authorized attribute on an executable program file. It behaves the same as removing the file from an APF-authorized library and raises `PermissionError` exception when the operation fails.

Examples

```
import zos_util
import tempfile
f = tempfile.NamedTemporaryFile()
# To specify a file with IBM-1047 code set
fpath = f.name
zos_util.chtag(fpath, 1047)

# To specify a file with ISO8859-1 code set
zos_util.chtag(fpath)
tag_info = zos_util.get_tag_info(fpath)
print(f"CCSID:{tag_info[0]}, TXT_FLAG:{tag_info[1]}")

# set to tag_mixed mode
zos_util.tag_mixed(fpath)
tag_info = zos_util.get_tag_info(fpath)
print(f"CCSID:{tag_info[0]}, TXT_FLAG:{tag_info[1]}")

# remove the tag from the file
zos_util.untag(fpath)
```

```
tag_info = zos_util.get_tag_info(fpath)
print(f"CCSID:{tag_info[0]}, TXT_FLAG:{tag_info[1]}")
```

The output is printed as follows:

```
CCSID:819, TXT_FLAG:True
CCSID:819, TXT_FLAG:False
CCSID:0, TXT_FLAG:False
```

Chapter 5. Information on using distutils module

Distutils is the primary way of building and distributing Python packages. For more information about distils, see https://docs.python.org/3/library/distutils.html.

Writing a module

You can use the typical layout for a Python package or module as follows:

```
README
LICENSE
setup.py
requirements.txt
src/
    module.py
    module.c
include/
    module.h
docs/
    conf.py
    index.rst
tests/
    test_module.py
```

The `setup.py` is the `makefile` equivalent for Python modules and it is often invoked through the following commands:

`python3 setup.py build`
: builds the package, but does not install it.

`python3 setup.py sdist`
: builds a source distributable tape archived file of the package and contains all the source of your modules.

`python3 setup.py bdist`
: builds a binary distributable tape archived file of the package and contains only object files of your compiled code.

`python3 setup.py install`
: installs the package to <python install location>/lib/site-packages/<your package here>.

`python3 setup.py check`
: checks the package for correctness.

Distutils by default uses the compiler located at `/bin/xlc` to compile C source files. If you have set the environment variable *CC*, the compiler defined by the CC variable is used instead. If the Python package requires a C++ compiler, `/bin/xlc++` is used as by default unless the *CXX* environment variable is set, in which case the compiler defined by the CXX variable is used. Similarly, `/bin/xlc` is used as the default linker for both shared and static libraries. If *LD* or *LDSHARED* are set, *LD* and *LDSHARED,* are used for each library type respectively.

The usage of xlclang and xlclang++ is also supported. You can export `CC=<path to xlclang>` and `CXX=<path to xlclang++>` to enable xlclang or xlclang++.

Note: There might be compatibility issues when mixing xlc and xlclang for compiled code and thus only one should be used consistently for building and linking modules.

On z/OS, DLL (.dll) and shared object (.so) files require a special file called a definition side-deck. The side-deck describes the functions and the variables that can be imported from a DLL by the binder. These files are generated automatically by the compiler when creating a DLL or shared object. For more information about side-decks, see Binding z/OS XL C/C++ programs in z/OS XL C/C++ User's Guide.

Side-deck considerations: `Python.x` is included by default and for other libraries, distutils automatically attempts to find the relevant side-decks. However, side-decks can be explicitly added to the build by using the **`extra_compile_args`** parameter to the Extension Class in `setup.py`.

Note: By default, distutils automatically supplies compilation and linking parameters for python header files and libpython side-decks.

Note: If you use a dynamic library for Python packages, you should ensure that all `.so` or `.dll` files are found in your `LIBPATH`.

Troubleshooting

For more information about errors using distutils, see "Errors using distutils" on page 28.

Examples

A simple `setup.py` for a pure Python module is as follows:

```
from distutils.core import setup
setup(name='example',
   version='1.0',
   description='An example package for distutils',
   author='John Doe',
   author_email='john.doe@ibm.com',
   url='https://www.ibm.com',
   packages=["ibm_example"],
   )
```

The corresponding file layout would be as follows:

```
example/
   setup.py
   ibm_example/
      __init__.py
```

If you want to add a C source file to the module, you can do it with the following lines:

```
from distutils.core import setup
setup(name='example',
   version='1.0',
   description='An example package for distutils',
   author='John Doe',
   author_email='john.doe@ibm.com',
   url='https://www.ibm.com',
   packages=["ibm_example"],
   ext_modules=[Extension('foo', ['src/foo1.c', 'src/foo2.c'], include_dirs=['include'])]
   )
```

The file layout would look something as follows:

```
example/
    setup.py
ibm_example/
    __init__.py
include/
    foo.h
src/
    foo1.c
    foo2.c
```

Note: You can also add C++ files in an analogous manner. Make sure that you use the appropriate file extensions, since this is how Python determines which compiler to invoke for the source files. If you include several modules that are specified with different extensions, a separate shared library is produced per extension.

A `setup.py` example with an explicit side-deck is as follows.

```
from distutils.core import setup
setup(name='example',
   version='1.0',
   description='An example package for distutils',
   author='John Doe',
   author_email='john.doe@ibm.com',
   url='https://www.ibm.com',
   packages=["ibm_example"],
```

```
    ext_modules=[Extension('foo', ['src/foo1.c', 'src/foo2.c'], include_dirs=['include'],
extra_compile_args=[/usr/lib/example.x])]
    )
```

If your module requires the use of `dll` or `.so` files, distutils automatically attempts to find them. When the side-deck is in a non-standard location, you should modify your `setup.py` to include the side-deck with `extra_compile_args` as shown above.

Best practice

When writing modules for Python, you should consider external dependencies, which can be located in non-typical locations, or in locations that are platform-dependent. To alleviate the non-typical locations issue, you can create a `setup.cfg` file that allows you to specify values at installation time. For more information on `setup.cfg` files, see https://docs.python.org/3.8/distutils/configfile.html. For more information on how to extend Python with C or C++, see https://docs.python.org/3/c-api/index.html.

Chapter 6. Codesets and translation

All text that exists in the Python interpreter is represented as UTF-8. Support for explicit conversion of the text in IBM Open Enterprise Python for z/OS is enabled through both the built-in codecs library and the provided EBCDIC package. Additional information about the codecs module can be found at https://docs.python.org/3/library/codecs.html.

Both IBM-1047 and ASCII source files are supported. It is recommended that you tag all source files with their correct encodings. If a file is untagged, IBM Open Enterprise Python for z/OS attempts to automatically check the encoding and run the source file. If a file is tagged, Python attempts to decode it by using the tagged encoding. You should note that while the source file might be EBCDIC, all I/O continues to be in UTF-8 unless explicit conversions are performed.

New in IBM Open Enterprise Python for z/OS 3.8.0.3

By default, IBM Open Enterprise Python for z/OS performs conversion to UTF-8 on all I/O, even in binary mode. This allows the execution of most existing code that is not tag- or encoding-aware. However, in situations, where an unconverted byte stream is desired, for example, consuming binary data and using checksums to verify content, setting the environment variable PYTHON_BINARY_CVT to OFF will disable auto-conversion of files opened in binary mode, for example, files with flag 'rb', 'wb', or 'ab'. This matches the behavior of community CPython. Note that this may be problematic on z/OS when processing and relying upon tagged files or when UTF-8 is expected.

Note: Setting this flag also disables the tagging of all files written in binary mode and may alter the behavior of the existing code.

For more information about supported codesets, see "Supported codesets" on page 15. For more information about tagging behaviors, see "Tagging behaviors" on page 19.

Examples

To open, read, and write from or to an IBM-1047 file, use the following commands:

```
>>> f = open('./test', mode='w+', encoding='cp1047')
>>> lines = f.readlines()
>>> f.write('hello world')
>>> for line in lines:
. . .       f.write(line)
>>> f.close()
```

To print to stdout with IBM1047, use the following commands:

```
>>> s = "Hello World".encode("cp1047") # this converts our internally UTF-8 string into a bytes
object with the ebcdic character values
>>> print(s)
b'\xc8\x85\x93\x93\x96@\xe6\x96\x99\x93\x84'
```

To print to stdout with the EBCDIC package, use the following commands beginning with the import:

```
>>> import ebcdic
>>> s = "hello world".encode('cp1047')
>>> print(s)
b'\xc8\x85\x93\x93\x96@\xe6\x96\x99\x93\x84'
```

Supported codesets

This table lists the supported Coded Character Set Identifiers (CCSIDs) that are defined.

Table 1. Supported codesets for IBM Open Enterprise Python for z/OS

CCSID	Encoding	Alias'	Languages supported
819	ascii	646, us-ascii	English
947	big5	big5-tw, csbig5	Traditional Chinese
	big5hkscs	big5-hkscs, hkscs	Traditional Chinese
037	cp037	IBM037, IBM039	English
273	cp273		German
290	cp290		Japanese Katakana
424	cp424	EBCDIC-CP-HE, IBM424	Hebrew
437	cp437	437, IBM437	English
500	cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
720	cp720		Arabic
737	cp737		Greek
775	cp775	IBM775	Baltic languages
838	cp838		
850	cp850	850, IBM850	Western Europe
852	cp852	852, IBM852	Central and Eastern Europe
855	cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
856	cp856		Hebrew
857	cp857	857, IBM857	Turkish
860	cp860	860, IBM860	Portuguese
861	cp861	861, CP-IS, IBM861	Icelandic
862	cp862	862, IBM862	Hebrew
863	cp863	863, IBM863	Canadian
864	cp864	IBM864	Arabic
865	cp865	865, IBM865	Danish, Norwegian
866	cp866	866, IBM866	Russian
869	cp869	869, CP-GR, IBM869	Greek
874	cp874		Thai
875	cp875		Greek
932	cp932	932, ms932, mskanji, ms-kanji	Japanese
949	cp949	949, ms949, uhc	Korean
950	cp950	950, ms950	Traditional Chinese

Table 1. Supported codesets for IBM Open Enterprise Python for z/OS (continued)			
CCSID	**Encoding**	**Alias'**	**Languages supported**
1006	cp1006		Urdu
1026	cp1026	1026, ibm1026	Turkish
1047	cp1047		Western Europe
1097	cp1097		Farsi
1125	cp1125	1125,ibm1125,cp866u,ruscii	Ukrainian and Belarusian
1140	cp1140		Western Europe
1141	cp1141		Western Europe
1142	cp1142		Danish-Norwegian
1143	cp1143		Finnish-Swedish
1144	cp1144		Italian
1145	cp1145		Spanish
1146	cp1146		English(UK)
1147	cp1147		French
1148	cp1148		Western Europe
1149	cp1149		Icelandic
1250	cp1250	windows-1250	Central and Eastern Europe
1251	cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
1252	cp1252	windows-1252	Western Europe
1253	cp1253	windows-1253	Greek
1254	cp1254	windows-1254	Turkish
1255	cp1255	windows-1255	Hebrew
1256	cp1256	windows-1256	Arabic
1257	cp1257	windows-1257	Baltic languages
1258	cp1258	windows-1258	Vietnamese
1350	euc_jp	eucjp, ujis, u-jis	Japanese
9582	euc_jis_2004	jisx0213, eucjis2004	Japanese
9591	euc_jisx0213	eucjisx0213	Japanese
971	euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean

CCSID	Encoding	Alias'	Languages supported
	gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
936	gbk	936, cp936, ms936	Unified Chinese
9444	gb18030	gb18030-2000	Unified Chinese
	hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
17336	iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
17337	iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
	iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
	iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
	iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
	iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
	iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
819	latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
25488	iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
	iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
	iso8859_4	iso-8859-4, latin4, L4	Baltic languages
25491	iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
	iso8859_6	iso-8859-6, arabic	Arabic
	iso8859_7	iso-8859-7, greek, greek8	Greek
	iso8859_8	iso-8859-8, hebrew	Hebrew
	iso8859_9	iso-8859-9, latin5, L5	Turkish
	iso8859_10	iso-8859-10, latin6, L6	Nordic languages
	iso8859_13	iso-8859-13	Baltic languages
	iso8859_14	iso-8859-14, latin8, L8	Celtic languages

Table 1. Supported codesets for IBM Open Enterprise Python for z/OS (continued)

Table 1. Supported codesets for IBM Open Enterprise Python for z/OS (continued)

CCSID	Encoding	Alias'	Languages supported
	iso8859_15	iso-8859-15	Western Europe
	johab	cp1361, ms1361	Korean
1167	koi8_r		Russian
1168	koi8_u		Ukrainian
1283	mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
1280	mac_greek	macgreek	Greek
1286	mac_iceland	maciceland	Icelandic
1282	mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
1285	mac_roman	macroman	Western Europe
1281	mac_turkish	macturkish	Turkish
	ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
	shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
	shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
1393	shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
	utf_16	U16, utf16	all languages
13489	utf_16_be	UTF-16BE	all languages (BMP only)
13491	utf_16_le	UTF-16LE	all languages (BMP only)
	utf_7	U7	all languages
13497	utf_8	U8, UTF, utf8	all languages

Tagging behaviors

File tags are used to identify the code set (encoding) of text data within files. IBM Open Enterprise Python for z/OS supports auto tagging files opened by using the open built-in. Below is a table that enumerates the behavior of file tags after Python I/O. This pattern is a special case for both UTF-8 and CP1047. Support for file tags with other encodings is enabled by the zos_util package. For more information about zos_util package, see Chapter 4, "Package documentation for zos_util," on page 9 and for more information about z/OS file tags, see File tagging in Enhanced ASCII section in z/OS UNIX System Services User's Guide.

Table 2. File tags for open built-in function

File name	Tag (before I/O)	Specified encoding	Resulting tag (after I/O)
test_file_1	(none)	cp1047	cp1047

Table 2. File tags for open built-in function (continued)

File name	**Tag (before I/O)**	**Specified encoding**	**Resulting tag (after I/O)**
test_file_1	iso8859-1	cp1047	cp1047
test_file_1	cp1047	cp1047	cp1047
test_file_2	(none)	(none)	iso8859-1
test_file_2	iso8859-1	(none)	iso8859-1
test_file_2	cp1047	(none)	cp1047
test_file_3	(none)	utf8	iso8859-1
test_file_3	iso8859-1	utf8	iso8859-1
test_file_3	cp1047	utf8	iso8859-1

Chapter 7. Virtual environments and considerations

IBM Open Enterprise Python for z/OS provides the `venv` module for creating lightweight virtual environments. This module allows you to manage separate package installations for different projects. To create a virtual environment, run the `venv` module as a script with the directory path as the following command:

```
python3 -m venv  /path/to/new/virtual/environment
```

The previous command creates a target directory and a bin subdirectory that contains a copy of the Python binaries files and link to standard libraries. If you want to pull all packages bundled with IBM Open Enterprise Python for z/OS into the virtual environments, run the above command with `--system-site-packages` option:

```
python3 -m venv  /path/to/new/virtual/environment --system-site-packages
```

The previous command creates the virtual environments that contain all the IBM Open Enterprise Python bundled packages such as: Numpy, cffi, cryptography, zos_util, and so on.

IBM Open Enterprise Python for z/OS contains additional packages for compatibility with z/OS. These packages come in two groups:

1. Packages that contains additional features to interact with z/OS Unix System Services, such as file tagging and EBCDIC encodings.
2. Prebuilt PyPI packages. By default, creating a virtual environment creates a clean environment, which means no packages installed. Specifying the `--system-site-packages` flag exposes these additional packages contained within IBM Open Enterprise Python for z/OS, so that they can be used within your virtual environment. This action is required if you need to install a package that has dependencies on one of these bundled packages. Otherwise, pip installs packages from PyPI which can lead to installation failure.

Once you create a virtual environment, you can activate it by running the following line:

```
. </path/to/new/virtual/environment/>/bin/activate
```

For more information about installing packaging by using `pip` and virtual environments, see https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/ and https://docs.python.org/3.8/library/venv.html?highlight=venv#module-venv.

Chapter 8. Debugging

You can debug an IBM Open Enterprise Python for z/OS application by using the built-in source code debugger via the pdb module.

The pdb module is part of the Python standard library and can be used to set conditional breakpoints, expression evaluation, view stack frames, and step through the code line by line. The code below shows an example where a breakpoint is set inside a **for** loop.

```
import pdb
for i in range(10):
    pdb.set_trace()
    i_square = i * i
    print("The square of {} is: {}".format(i, i_square))
```

You don't necessarily need to import pdb in an application to debug. You can invoke the debugger on a script by calling it from the command line. Create a new file called `pdb_debugger.py` and add the following lines:

```
import os, sys

SCRIPT_DIR, SCRIPT_NAME = os.path.split(os.path.abspath(__file__))
PARENT_DIR = os.path.dirname(SCRIPT_DIR)
filename = __file__

def func_a():
    x = 5
    y = 15

    z = x + y

    return z

def func_b():

    sum = 0
    for i in range(5):
        sum = sum + i

    return sum

if __name__ == '__main__':
    print("Running {}".format(SCRIPT_NAME))

    z_ret = func_a()

    sum_ret = func_b()

    print("z_ret: {} \t\t sum_ret: {}".format(z_ret, sum_ret))
```

To invoke the debugger, run the script with the following command:

```
python3 -m pdb pdb_debugger.py
```

You can see the debug prompt at the first line of the file and you can then step through the code by using the commands outline in the pdb docs. For example, to break `func_b` when it is called, execute the following command in the prompt:

```
break func_b
```

This command registers the breakpoint at that function call. Start the file execution by inputting the command:

```
c
```

This command now starts executing the code line by line and stops the execution just before the first line of the `func_b` function.

Alternatively, you can achieve the same by passing additional parameters when invoking the debugger to run the script:

```
python3 -m pdb -c "break func_b" pdb_debugger.py
```

More information about the capabilities and documentation of the pdb module can be found here.

Python also includes the `trace` module, which can be used to monitor functions and line execution. You can use the useful features that are provided by the `trace` module such as code-coverage and you can run the module either from the command prompt or incorporate the module into the program itself.

The example below gives a brief overview of the capabilities of using the `trace` module.

Since the `trace` module can create associated files that hold the trace results, create a subfolder that houses the code to be run:

```
$ mkdir -p fibonacci
```

Create the files with the following code:

* `fibonacci/fibonacci.py`

```
"""
In mathematics, the Fibonacci numbers, form a sequence, called the Fibonacci sequence, such
that each number is the
sum of the two preceding ones, starting from F(0) = 0 and F(1) = 1, and for any integer n > 1:
F(n) = F(n-1) + F(n-2).
[wikipedia](https://en.wikipedia.org/wiki/Fibonacci_number)
"""
def fib(x):
    print("Processing fib({})".format(x))
    if x<0:
        print("Input needs to be a positive integer")
    elif x==1:
        return 0
    elif x==2:
        return 1
    else:
        return fib(x-1)+fib(x-2)
if __name__ == '__main__':
    print(fib(15))
```

* `fibonacci/main.py`

```
from fibonacci import fib
def main():
    print ("In the main program.")
    fib(4)
    return
if __name__ == '__main__':
    main()
```

* `fibonacci_trace.py`

```
import trace
from fibonacci.fibonacci import fib
tracer = trace.Trace(count=False, trace=True)
tracer.run('fib(4)')
```

To see which statements are being executed as the program runs, run the command:

```
python -m trace --trace fibonacci/main.py
```

To run the code coverage to see which lines are run and which are skipped, use the `--count` option:

```
python -m trace --count fibonacci/main.py
```

To see relationships between function calls, run the command:

```
python -m trace --listfuncs fibonacci/main.py
```

For more details, run the command:

```
python -m trace --listfuncs --trackcalls fibonacci/main.py
```

To invoke the trace from a python script, run the command:

```
python fibonacci_trace.py
```

You can learn more about the module and its full set of capabilities here.

Chapter 9. Troubleshooting

This chapter describes some common issues that you might encounter while creating your IBM Open Enterprise Python for z/OS applications.

For more troubleshooting information about multiprocessing considerations, see "Multiprocessing considerations" on page 29.

For more troubleshooting information about tagging files, see "Tagging files" on page 30.

Fatal Python Error: Failed to get random number

```
Fatal Python error: _Py_HashRandomization_Init: failed to get random numbers to initialize
Python
```

The above error shows up when you try to run Python without enabling Integrated Cryptographic Services Facility (ICSF). ICSF is typically responsible for supplying random data for `/dev/urandom`. You can run the following command to verify whether ICSF is enabled or not on your system.

```
head -c10 /dev/urandom
```

If ICSF is enabled, you will see random data and if it is not enabled on your system, you will encounter `Internal Error`.

For more information including installation guide, please refer to ICSF System Programmer's Guide (SC14-7507) and ICSF Administrator's Guide (SC14-7506).

Python execution failure due to semaphore exhaustion

When semaphore exhaustion occurs on a machine, it can cause Python programs to fail in various ways. One common example will be an error message as follows:

```
_multiprocessing.SemLock(PermissionError: [Errno 139] EDC5139I Operation not permitted.
```

To diagnose whether semaphore exhaustion is an issue, you can run the following command to examine the number of semaphores that your user is currently using.

```
ipcs | grep <your ID>
```

If you run `ipcs -y`, you can get the limits for semaphores or the shared memory. If the number reported is close to that number, then it's likely that you are hitting the limit when running Python.

The following example shows how to approach cleaning up semaphore usage under your user id:

```
ipcs | grep <your ID> | awk '{print $2}' > semaphores_example.txt
for i in $(cat semaphores_example.txt) ; do { ipcrm -s $i >> /dev/null 2>&1; }; done;
for i in $(cat semaphores_example.txt) ; do { ipcrm -m $i >> /dev/null 2>&1; }; done;
rm semaphores_example.txt
```

Encodings module not found error

You might run a Python script and get `encodings` module not found errors. The presence of the *PYTHONHOME* environment variable can lead to the mixing of Python versions:

```
$ python3
Fatal Python error: initfsencoding: unable to load the file system codec
ModuleNotFoundError: No module named 'encodings'

Current thread 0x26b7980000000001 (most recent call first):
```

```
CEE5207E The signal SIGABRT was received.
ABORT instruction
```

This error is generally caused by setting the *PYTHONHOME* environment variable to a conflicting location. Try to set the *PYTHONHOME* environment variable and execute Python again using the following commands:

```
$ unset PYTHONHOME
$ python3
```

Errors using distutils

- If you see the following errors,

  ```
  FSUM3010 Specify a file with the correct suffix (.C, .hh, .i, .c, .i, .s, .o, .x, .p, .I,
  or .a), or a corresponding data set name....
  ```

  export the following environment variables:

  ```
  export _CC_CCMODE=1
  export _CXX_CCMODE=1
  export _C89_CCMODE=1
  export _CC_EXTRA_ARGS=1
  export _CXX_EXTRA_ARGS=1
  export _C89_EXTRA_ARGS=1
  ```

- If you see the following warnings,

  ```
  WARNING CCN3236 /usr/include/unistd.h:1169  Macro name _POSIX_THREADS has been redefined.
  ```

  you can safely ignore as Python forces `POSIX` thread behavior in modules for compatibility reasons.

  If you see the similar error with xlc:

  ```
  "/usr/include/unistd.h", line 1169.16: CCN5848 (S) The macro name "_POSIX_THREADS" is already
  defined with a different definition.
  ```

  then setting the appropriate qlanglvl with redefmac can be used to work around this.

  For both xlclang and xlc, if non-POSIX thread behavior is required, you might undefine this macro in your source files. This action should be done with care and is not recommended.

- If you see the following error while trying to install a package:

  ```
  error: [Errno 129] EDC5129I No such file or directory.: '/bin/xlc'
  ```

  This means the package that you are attempting to install requires a C compiler. If you have one in a non-standard location, you can specify it with the following command:

  ```
  CC=<path to C compiler>
  CXX=<path to C++ compiler>
  ```

  If you do not have a C/C++ compiler installed on your system, you can acquire xlc or xlclang here.

Extended precision floating point support issue in NumPy library

The NumPy library in IBM Open Enterprise Python for z/OS does not support direct string conversion to the `long double` data type. Instead, literals and strings are parsed to a double precision floating point number followed by a conversion to the long double number. This indirect conversion introduces precision and range problems for numbers outside of the double precision range.

Incorrectly tagged files

If you see an error as follows,

```
SyntaxError: Non-UTF-8 code starting with '\x84' in file test.py on line 1, but no encoding
declared; see http://python.org/dev/peps/pep-0263/ for details
```

ensure that the file is either encoded as ASCII or IBM-1047 and correctly tagged. The interpreter doesn't attempt to auto-detect the file encoding if the file is already tagged. If the file is correctly tagged and encoded, check the non-printable characters in the file.

NumPy exec_command return value issue

The Numpy exec_command function of Numpy can be used to execute shell commands. Numpy uses the subprocess module to execute commands by using the default shell `/bin/sh`. While executing these commands, only the user-provided environment variables are passed to the subprocess. On z/OS, if the *_BPXK_AUTOCVT* environment variable is not set to `ON`, the default output from terminal commands might be in `EBCDIC`, which causes a mismatch error since the return value is expected to be in `ASCII`. To avoid this issue, you should follow the command line below to set an environment variable in the subprocess to get the return value in `ASCII`.

```
_BPXK_AUTOCVT='ON'
```

Packages unable to install cffi within a virtual environment

```
c/_cffi_backend.c:15:10: fatal error: 'ffi.h' file not found
```

The above error shows up when you try to install a package and require ffi.h file into a virtual environment without using site-packages. IBM Open Enterprise Python for z/OS is distributed with several packages installed, including cffi. To get access when using a virtual environment, create the environment with the flag `--system-site-packages` as the following example:

```
<path to python3 install>/bin/python3 -m venv --system-site-packages venv
```

Redirecting to a file in a shell script results in garbled output

If you are redirecting to a file in a shell script and notice that the Python output is garbled, set the following environment variables:

```
export _TAG_REDIR_ERR=txt
export _TAG_REDIR_IN=txt
export _TAG_REDIR_OUT=txt
```

UnicodeDecodeError error message

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa3 in position 0: invalid start byte
```

This error is returned when there is a conversion error. This error message usually refers to a file that is opened is tagged incorrectly, or has a wrong encoding specified. To verify whether the encodings are correct, see Chapter 6, “Codesets and translation,” on page 15 for more details.

Multiprocessing considerations

IBM Open Enterprise Python for z/OS sets the default method for creating new processes to spawn. Fork is known to cause crashes of subprocesses in a multithreaded context and is the default used in most Unix systems. If you port an application from a Unix system, you might need to make some changes to your codebase to make it compatible. For a description of the differences between the fork and the spawn, and potentially any changes that are required, see https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods.

If you use fork, the most common symptom is getting a runtime error as follows:

```
RuntimeError: can't start new thread
```

Tagging files

IBM Open Enterprise Python for z/OS supports both EBCDIC and ASCII input files. It attempts to autodetect file encodings, but it is highly recommended that all source files be tagged with their correct encodings.

You can use the chtag utility to tag input files that are not EBCDIC text, which is the default encoding for input files on z/OS.

Binary files

To tag a file as binary, use the following command:

```
chtag -b <path/to/binary/file>
```

To verify if the file has the binary tag, use the following command:

```
ls -T <path/to/binary/file>
```

You get the following output:

```
b binary T=off path/to/binary/file
```

Enhanced ASCII support

Some applications take advantage of Enhanced ASCII support, which requires ASCII encoded text files to be tagged as ASCII text files. Python applications on z/OS also support reading files that are tagged as ASCII text files.

To tag a file as an ASCII text file, use the following command:

```
chtag -tc ISO8859-1 <path/to/ascii/file>
```

To verify that a file is tagged as an ASCII text file, use the following command:

```
ls -T <path/to/ascii/file>
```

You get the following output:

```
t ISO8859-1   T=on  path/to/ascii/file
```

Usage

If you have some source files on an ASCII platform and you want to use them on z/OS, you can tag those files with the following steps:

1. Create a zip file of your source files on the ASCII platform.
2. Unzip the zip file on z/OS.
3. Tag all text files by using the following command:

   ```
   chtag -tc ISO8859-1
   ```

4. Tag all binary files by using the following command:

   ```
   chtag -b
   ```

To copy files remotely from an ASCII platform to z/OS, you can use the `sftp` command, which converts every file from ASCII to EBCDIC as it copies. In this case, tagging is not necessary.

If you are redirecting to a file in a shell script, set the following environment variables; otherwise you get garbled output.

```
export _TAG_REDIR_ERR=txt
export _TAG_REDIR_IN=txt
export _TAG_REDIR_OUT=txt
```

Troubleshooting

For more information about troubleshooting incorrectly tagged files, see "Incorrectly tagged files" on page 28.

Chapter 10. Support, best practices, and resources

This section lists IBM Open Enterprise Python for z/OS support, best practices, and learning resources.

- "Support" on page 33
- "Best practices" on page 34
- "Learning resources" on page 36

Support

To find help about IBM Open Enterprise Python for z/OS, it is important to collect as much information as possible about your installation configuration.

To establish what version of IBM Open Enterprise Python for z/OS is in use, run the following command:

```
python3 --version
```

The version of Python is displayed.

To get more details about the exact build of the IBM Open Enterprise Python for z/OS, run the following command:

```
python3 -c “import sys; print(sys.version)”
```

This prints additional information about Python.

IBM Open Enterprise Python for z/OS includes the python pip utility for working with modules and packages. To establish what version of pip is in use, run the following command:

```
pip3 --version
```

The version of the pip utility is displayed.

To get more details about the libraries shipped with IBM Open Enterprise Python for z/OS, run the following command:

```
pip3 list
```

The versions of all installed libraries will be displayed.

Note: Only cffi, cryptography, ebcdic, numpy, pip, pycparser, setuptools, six and setuptools are officially supported by IBM Open Enterprise Python for z/OS.

Online self-help

Online documentation is available on this Knowledge Center. You can also download the PDF format documentation for offline use.

Getting IBM experts to solve your problem by opening a case

Paid IBM Subscription & Support (S&S) entitles you to world-class IBM support for IBM Open Enterprise Python for z/OS. Get IBM support by opening a case. First, obtain the SMP/E edition and purchase S&S. Once you have purchased S&S, open a case after logging in with your IBM customer ID to request support from IBM. If you need help on opening a case, see IBM Support page.

General Help

For help with writing Python programs, working with the standard library or other general inquiries, see https://docs.python.org/3.8/.

Best practices

Virtual environments

When you install packages via pip, you might want to create a virtual environment to isolate package installation from the global installation directory. You can create a virtual environment with the following command.

```
python3 -m venv <name of venv>
```

After you create a virtual environment, you can **activate** it by sourcing the script that is located in <name of venv>/bin/activate and then deactivate it with the command **deactivate**. You can verify your current venv by checking your shell prompt that should now be prefixed with <name of venv>. Once using a venv, all pip installed packages are placed in <name of venv>/lib/python3.8/site-packages. You can reference the commands as follows:

```
bash-4.3$ python3 -m venv my_venv
bash-4.3$ . my_venv/bin/activate
(my_venv) bash-4.3$ pip3 install <package>         # this will install into the venv
(my_venv) bash-4.3$ deactivate                     # this will deactivate the venv
$                                                  # note the shell prompt is no longer prefixed
```

Note: If you want to use any of the bundled packages such as Numpy in your venv, you must add the option --system-site-packages, to verify that these packages are in your venv by simply running the following commands:

```
(my_venv) bash-4.3$ pip3 list
Package    Version
----------    -------
numpy       1.18.2
```

For more information on virtual environments, refer to the CPython documentation at https://docs.python.org/3/library/venv.html#venv-def.

Security and pip

Pip is a tool that connects to the internet and executes setup files. It is advised that you **never** run pip as a privileged user or with any elevated permissions, since pip runs arbitrary code to install packages. Additionally, if you run pip as an elevated user, you might inadvertently globally install packages, which can alter the intended behavior of Python for all users.

Multiprocessing

Multiprocessing enables side-stepping of the Global Interpreter Lock in CPython, which is useful not only for task parallelization, but also for preventing long-lived tasks, for example, handling https requests from blocking the main flow of the program. Multiprocessing is available in Python as the multiprocessing package, where processes can be created by using the Pool class, which offers a convenient way for setting up the parallel processing and the Process class. It is useful for controlling individual processes.

Here is a multiprocessing example with the Pool class:

```
from multiprocessing import Pool

def times_two(x):
    return 2 * x

if __name__ == "__main__":
    pool = Pool(100)
    print(pool.map(times_two, [x for x in range(1000)]))    #[0, 2, 4, … , 1998]
```

Here is a multiprocessing example with the `Process` class:

```
from multiprocessing import Process
import time

def long_process(seconds):
    print("long_process started.")
    time.sleep(int(seconds))
    print("long_process finished.")


if __name__ == "__main__":
    proc = Process(target=long_process, args=('10',))
    proc.start()
    print("Hi from the main process.")
    proc.join()
```

For further information on multiprocessing, see official Python reference.

Unit testing and code coverage

Python includes two separate but similar tools to aid you to test your code. The first tool is unittest, which is used to create and run distinct unit tests:

```
import unittest
def test_func(a, b):
    return a + b
class ExampleTest(unittest.TestCase):
    def test_add(self):
    self.assertEqual(test_func(1,2), 3)
if __name__ == '__main__':
    unittest.main()
```

You can run the above example with the following commands:

```
python3 <filename.py>
python3 -m unittest <filename.py>
```

More information about the built-in unit testing framework can be found here.

The second module that Python includes for testing is doctest. Instead of having separate unittest files, doctest is used for inline testing directly into the function itself. It is useful for testing pure functions, and not meant to replace tradition unit testing:

```
def test_func(a, b):
    '''
    Return the sum of a + b
    >>> test_func(5, 5)
    10
    >>> test_func(5.0, 5.0)
    10.0
    '''
    return a + b
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

You can run the above example with the following commands:

```
python3 -m doctest -v <filename.py>
python3 <filename.py>
```

More information about doctest can be found here.

In addition to unit testing, code coverage tooling can assist with determining what hasn't been tested by showing which lines haven't been run by the interpreter. While there are no built-in code coverage tools in a Python distribution, PyPI has multiple code coverage frameworks that integrate with both the built-in unit testing framework and other PyPI testing frameworks.

Learning resources

This topic lists both Python community resources and IBM resources that you can refer to.

Python community resources

- Python Community
- Official Python 3 Documentation
- Beginner's Guide to Python
- Python Developer's Guide
- PyVideo.org - Talks from various Python conferences
- The Python Package Index
- NumPy - A package for scientific computing

IBM resources

- IBM Community
- Redbooks® Introduction to z/OS
- IBM Online Software Catalog
- Explore IBM Systems
- RedHat Ansible® - A platform for IT automation
- Z Open Automation Utilities

Product Number: 5655-PYT

SC28-3143-00 (2020-06-26 updated)