os — Portable access to operating system specific features

Purpose: Portable access to operating system specific features.

The os module provides a wrapper for platform specific modules such as `posix`, `nt`, and `mac`. The API for functions available on all platforms should be the same, so using the os module offers some measure of portability. Not all functions are available on every platform, however. Many of the process management functions described in this summary are not available for Windows.

The Python documentation for the os module is subtitled “Miscellaneous operating system interfaces”. The module consists mostly of functions for creating and managing running processes or file system content (files and directories), with a few other bits of functionality thrown in besides.

Examining the File System Contents

To prepare a list of the contents of a directory on the file system, use `listdir()`.

```
# os_listdir.py

import os
import sys

print(sorted(os.listdir(sys.argv[1])))
```

The return value is an unsorted list of all of the named members of the directory given. No distinction is made between files, subdirectories, or symlinks.

```
$ python3 os_listdir.py .

['index.rst', 'os_access.py', 'os_cwd_example.py',
'os_directories.py', 'os_environ_example.py',
'os_exec_example.py', 'os_fork_example.py',
'os_kill_example.py', 'os_listdir.py', 'os_listdir.py~',
'os_process_id_example.py', 'os_process_user_example.py',
'os_rename_replace.py', 'os_rename_replace.py~',
'os_scandir.py', 'os_scandir.py~', 'os_spawn_example.py',
'os_stat.py', 'os_stat_chmod.py', 'os_stat_chmod_example.txt',
'os_strerror.py', 'os_strerror.py~', 'os_symlinks.py',
'os_system_background.py', 'os_system_example.py',
'os_system_shell.py', 'os_wait_example.py',
'os_waitpid_example.py', 'os_walk.py']
```

The function `walk()` traverses a directory recursively and for each subdirectory generates a `tuple` containing the directory path, any immediate sub-directories of that path, and a list of the names of any files in that directory.

```
# os_walk.py

import os
import sys

# If we are not given a path to list, use /tmp
if len(sys.argv) == 1:
    root = '/tmp'
else:
    root = sys.argv[1]

for dir_name, sub_dirs, files in os.walk(root):
    print(dir_name)
    # Make the subdirectory names stand out with /
    sub_dirs = [n + '/' for n in sub_dirs]
    # Mix the directory contents together
    contents = sub_dirs + files
    contents.sort()
    # Show the contents
    for c in contents:
        print('  {}'.format(c))
    print()
```

This example shows a recursive directory listing.

```
$ python3 os_walk.py ../zipimport

../zipimport
  __init__.py
  example_package/
  index.rst
  zipimport_example.zip
  zipimport_find_module.py
  zipimport_get_code.py
  zipimport_get_data.py
  zipimport_get_data_nozip.py
  zipimport_get_data_zip.py
  zipimport_get_source.py
  zipimport_is_package.py
  zipimport_load_module.py
  zipimport_make_example.py

../zipimport/example_package
  README.txt
  __init__.py
```

If more information is needed than the names of the files, it is likely to be more efficient to use `scandir()` than `listdir()` because more information is collected in one system call when the directory is scanned.

```
# os_scandir.py

import os
import sys

for entry in os.scandir(sys.argv[1]):
    if entry.is_dir():
        typ = 'dir'
    elif entry.is_file():
        typ = 'file'
    elif entry.is_symlink():
        typ = 'link'
    else:
        typ = 'unknown'
    print('{name} {typ}'.format(
        name=entry.name,
        typ=typ,
    ))
```

`scandir()` returns a sequence of `DirEntry` instances for the items in the directory. The object has several attributes and methods for accessing metadata about the file.

```
$ python3 os_scandir.py .

index.rst file
os_access.py file
os_cwd_example.py file
os_directories.py file
os_environ_example.py file
os_exec_example.py file
os_fork_example.py file
os_kill_example.py file
os_listdir.py file
os_listdir.py~ file
os_process_id_example.py file
os_process_user_example.py file
```

```
os_rename_replace.py file
os_rename_replace.py~ file
os_scandir.py file
os_scandir.py~ file
os_spawn_example.py file
os_stat.py file
os_stat_chmod.py file
os_stat_chmod_example.txt file
os_strerror.py file
os_strerror.py~ file
os_symlinks.py file
os_system_background.py file
os_system_example.py file
os_system_shell.py file
os_wait_example.py file
os_waitpid_example.py file
os_walk.py file
```

Managing File System Permissions

Detailed information about a file can be accessed using `stat()` or `lstat()` (for checking the status of something that might be a symbolic link).

```python
# os_stat.py

import os
import sys
import time

if len(sys.argv) == 1:
    filename = __file__
else:
    filename = sys.argv[1]

stat_info = os.stat(filename)

print('os.stat({}):'.format(filename))
print('  Size:', stat_info.st_size)
print('  Permissions:', oct(stat_info.st_mode))
print('  Owner:', stat_info.st_uid)
print('  Device:', stat_info.st_dev)
print('  Created      :', time.ctime(stat_info.st_ctime))
print('  Last modified:', time.ctime(stat_info.st_mtime))
print('  Last accessed:', time.ctime(stat_info.st_atime))
```

The output will vary depending on how the example code was installed. Try passing different filenames on the command line to `os_stat.py`.

```
$ python3 os_stat.py

os.stat(os_stat.py):
  Size: 593
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created      : Sat Dec 17 12:09:51 2016
  Last modified: Sat Dec 17 12:09:51 2016
  Last accessed: Sat Dec 31 12:33:19 2016

$ python3 os_stat.py index.rst

os.stat(index.rst):
  Size: 26878
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created      : Sat Dec 31 12:33:10 2016
  Last modified: Sat Dec 31 12:33:10 2016
  Last accessed: Sat Dec 31 12:33:19 2016
```

On Unix-like systems, file permissions can be changed using `chmod()`, passing the mode as an integer. Mode values can be constructed using constants defined in the `stat` module. This example toggles the user's execute permission bit:

```
# os_stat_chmod.py

import os
import stat

filename = 'os_stat_chmod_example.txt'
if os.path.exists(filename):
    os.unlink(filename)
with open(filename, 'wt') as f:
    f.write('contents')

# Determine what permissions are already set using stat
existing_permissions = stat.S_IMODE(os.stat(filename).st_mode)

if not os.access(filename, os.X_OK):
    print('Adding execute permission')
    new_permissions = existing_permissions | stat.S_IXUSR
else:
    print('Removing execute permission')
    # use xor to remove the user execute permission
    new_permissions = existing_permissions ^ stat.S_IXUSR

os.chmod(filename, new_permissions)
```

The script assumes it has the permissions necessary to modify the mode of the file when run.

```
$ python3 os_stat_chmod.py

Adding execute permission
```

The function `access()` can be used to test the access rights a process has for a file.

```
# os_access.py

import os

print('Testing:', __file__)
print('Exists:', os.access(__file__, os.F_OK))
print('Readable:', os.access(__file__, os.R_OK))
print('Writable:', os.access(__file__, os.W_OK))
print('Executable:', os.access(__file__, os.X_OK))
```

The results will vary depending on how the example code is installed, but the output will be similar to this:

```
$ python3 os_access.py

Testing: os_access.py
Exists: True
Readable: True
Writable: True
Executable: False
```

The library documentation for `access()` includes two special warnings. First, there is not much sense in calling `access()` to test whether a file can be opened before actually calling `open()` on it. There is a small, but real, window of time between the two calls during which the permissions on the file could change. The other warning applies mostly to networked file systems that extend the POSIX permission semantics. Some file system types may respond to the POSIX call that a process has permission to access a file, then report a failure when the attempt is made using `open()` for some reason not tested via the POSIX call. All in all, it is better to call `open()` with the required mode and catch the `IOError` raised if there is a problem.

Creating and Deleting Directories

There are several functions for working with directories on the file system, including creating, listing contents, and removing them.

```
# os_directories.py
```

```
import os

dir_name = 'os_directories_example'

print('Creating', dir_name)
os.makedirs(dir_name)

file_name = os.path.join(dir_name, 'example.txt')
print('Creating', file_name)
with open(file_name, 'wt') as f:
    f.write('example file')

print('Cleaning up')
os.unlink(file_name)
os.rmdir(dir_name)
```

There are two sets of functions for creating and deleting directories. When creating a new directory with `mkdir()`, all of the parent directories must already exist. When removing a directory with `rmdir()`, only the leaf directory (the last part of the path) is actually removed. In contrast, `makedirs()` and `removedirs()` operate on all of the nodes in the path. `makedirs()` will create any parts of the path that do not exist, and `removedirs()` will remove all of the parent directories, as long as they are empty.

```
$ python3 os_directories.py

Creating os_directories_example
Creating os_directories_example/example.txt
Cleaning up
```

Working with Symbolic Links

For platforms and file systems that support them, there are functions for working with symlinks.

```
# os_symlinks.py

import os

link_name = '/tmp/' + os.path.basename(__file__)

print('Creating link {} -> {}'.format(link_name, __file__))
os.symlink(__file__, link_name)

stat_info = os.lstat(link_name)
print('Permissions:', oct(stat_info.st_mode))

print('Points to:', os.readlink(link_name))

# Cleanup
os.unlink(link_name)
```

Use `symlink()` to create a symbolic link and `readlink()` for reading it to determine the original file pointed to by the link. The `lstat()` function is like `stat()`, but operates on symbolic links.

```
$ python3 os_symlinks.py

Creating link /tmp/os_symlinks.py -> os_symlinks.py
Permissions: 0o120755
Points to: os_symlinks.py
```

Safely Replacing an Existing File

Replacing or renaming an existing file is not idempotent and may expose applications to race conditions. The `rename()` and `replace()` functions implement safe algorithms for these actions, using atomic operations on POSIX-compliant systems when possible.

```
# os_rename_replace.py
```

```
import glob
import os


with open('rename_start.txt', 'w') as f:
    f.write('starting as rename_start.txt')

print('Starting:', glob.glob('rename*.txt'))

os.rename('rename_start.txt', 'rename_finish.txt')

print('After rename:', glob.glob('rename*.txt'))

with open('rename_finish.txt', 'r') as f:
    print('Contents:', repr(f.read()))

with open('rename_new_contents.txt', 'w') as f:
    f.write('ending with contents of rename_new_contents.txt')

os.replace('rename_new_contents.txt', 'rename_finish.txt')

with open('rename_finish.txt', 'r') as f:
    print('After replace:', repr(f.read()))

for name in glob.glob('rename*.txt'):
    os.unlink(name)
```

The rename() and replace() functions work across filesystems, most of the time. Renaming a file may fail if it is being moved to a new fileystem or if the destination already exists.

```
$ python3 os_rename_replace.py

Starting: ['rename_start.txt']
After rename: ['rename_finish.txt']
Contents: 'starting as rename_start.txt'
After replace: 'ending with contents of rename_new_contents.txt'
```

Detecting and Changing the Process Owner

The next set of functions provided by os are used for determining and changing the process owner ids. These are most frequently used by authors of daemons or special system programs that need to change permission level rather than running as root. This section does not try to explain all of the intricate details of Unix security, process owners, etc. See the references list at the end of this section for more details.

The following example shows the real and effective user and group information for a process, and then changes the effective values. This is similar to what a daemon would need to do when it starts as root during a system boot, to lower the privilege level and run as a different user.

Note

Before running the example, change the TEST_GID and TEST_UID values to match a real user defined on the system.

```
# os_process_user_example.py

import os

TEST_GID = 502
TEST_UID = 502


def show_user_info():
    print('User (actual/effective)  : {} / {}'.format(
        os.getuid(), os.geteuid()))
    print('Group (actual/effective) : {} / {}'.format(
        os.getgid(), os.getegid()))
    print('Actual Groups   :', os.getgroups())
print('BEFORE CHANGE:')
show_user_info()
print()

try:
    os.setegid(TEST_GID)
except OSError:
    print('ERROR: Could not change effective group. '
          'Rerun as root.')
else:
    print('CHANGE GROUP:')
    show_user_info()
    print()

try:
    os.seteuid(TEST_UID)
except OSError:
    print('ERROR: Could not change effective user. '
          'Rerun as root.')
else:
    print('CHANGE USER:')
    show_user_info()
    print()
```

When run as user with id of 502 and group 502 on OS X, this output is produced:

```
$ python3 os_process_user_example.py

BEFORE CHANGE:
User (actual/effective)  : 527 / 527
Group (actual/effective) : 501 / 501
Actual Groups   : [501, 701, 402, 702, 500, 12, 61, 80, 98, 398,
399, 33, 100, 204, 395]

ERROR: Could not change effective group. Rerun as root.
ERROR: Could not change effective user. Rerun as root.
```

The values do not change because when it is not running as root, a process cannot change its effective owner value. Any attempt to set the effective user id or group id to anything other than that of the current user causes an `OSError`. Running the same script using sudo so that it starts out with root privileges is a different story.

```
$ sudo python3 os_process_user_example.py

BEFORE CHANGE:

User (actual/effective)  : 0 / 0
Group (actual/effective) : 0 / 0
Actual Groups : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]

CHANGE GROUP:
User (actual/effective)  : 0 / 0
Group (actual/effective) : 0 / 502
Actual Groups   : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]

CHANGE USER:
User (actual/effective)  : 0 / 502
Group (actual/effective) : 0 / 502
Actual Groups   : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]
```

In this case, since it starts as root, the script can change the effective user and group for the process. Once the effective UID is changed, the process is limited to the permissions of that user. Because non-root users cannot change their effective group, the program needs to change the group before changing the user.

Managing the Process Environment

Another feature of the operating system exposed to a program though the os module is the environment. Variables set in the environment are visible as strings that can be read through `os.environ` or `getenv()`. Environment variables are commonly used for configuration values such as search paths, file locations, and debug flags. This example shows how to retrieve an environment variable, and pass a value through to a child process.

```python
# os_environ_example.py

import os

print('Initial value:', os.environ.get('TESTVAR', None))
print('Child process:')
os.system('echo $TESTVAR')

os.environ['TESTVAR'] = 'THIS VALUE WAS CHANGED'

print()
print('Changed value:', os.environ['TESTVAR'])
print('Child process:')
os.system('echo $TESTVAR')

del os.environ['TESTVAR']

print()
print('Removed value:', os.environ.get('TESTVAR', None))
print('Child process:')
os.system('echo $TESTVAR')
```

The `os.environ` object follows the standard Python mapping API for retrieving and setting values. Changes to `os.environ` are exported for child processes.

```
$ python3 -u os_environ_example.py

Initial value: None
Child process:


Changed value: THIS VALUE WAS CHANGED
Child process:
THIS VALUE WAS CHANGED

Removed value: None
Child process:
```

Managing the Process Working Directory

Operating systems with hierarchical file systems have a concept of the *current working directory* – the directory on the file system the process uses as the starting location when files are accessed with relative paths. The current working directory can be retrieved with `getcwd()` and changed with `chdir()`.

```python
# os_cwd_example.py

import os

print('Starting:', os.getcwd())

print('Moving up one:', os.pardir)
os.chdir(os.pardir)

print('After move:', os.getcwd())
```

`os.curdir` and `os.pardir` are used to refer to the current and parent directories in a portable manner.

```
$ python3 os_cwd_example.py

Starting: .../pymotw-3/source/os
Moving up one: ..
After move: .../pymotw-3/source
```

Running External Commands

Warning

Many of these functions for working with processes have limited portability. For a more consistent way to work with processes in a platform independent manner, see the `subprocess` module instead.

The most basic way to run a separate command, without interacting with it at all, is `system()`. It takes a single string argument, which is the command line to be executed by a sub-process running a shell.

```
# os_system_example.py

import os

# Simple command
os.system('pwd')
```

The return value of `system()` is the exit value of the shell running the program packed into a 16 bit number, with the high byte the exit status and the low byte the signal number that caused the process to die, or zero.

```
$ python3 -u os_system_example.py

.../pymotw-3/source/os
```

Since the command is passed directly to the shell for processing, it can include shell syntax such as globbing or environment variables.

```
# os_system_shell.py

import os

# Command with shell expansion
os.system('echo $TMPDIR')
```

The environment variable `$TMPDIR` in this string is expanded when the shell runs the command line.

```
$ python3 -u os_system_shell.py

/var/folders/5q/8gk0wq888xlggz008k8dr7180000hg/T/
```

Unless the command is explicitly run in the background, the call to `system()` blocks until it is complete. Standard input, output, and error from the child process are tied to the appropriate streams owned by the caller by default, but can be redirected using shell syntax.

```
# os_system_background.py

import os
import time

print('Calling...')
os.system('date; (sleep 3; date) &')

print('Sleeping...')
time.sleep(5)
```

This is getting into shell trickery, though, and there are better ways to accomplish the same thing.

```
$ python3 -u os_system_background.py

Calling...
Sat Dec 31 12:33:20 EST 2016
Sleeping...
Sat Dec 31 12:33:23 EST 2016
```

Creating Processes with os.fork()

The POSIX functions `fork()` and `exec()` (available under Mac OS X, Linux, and other Unix variants) are exposed via the `os` module. Entire books have been written about reliably using these functions, so check the library or bookstore for more details than are presented here in this introduction.

To create a new process as a clone of the current process, use `fork()`:

```
# os_fork_example.py

import os

pid = os.fork()

if pid:
    print('Child process id:', pid)
else:
    print('I am the child')
```

The output will vary based on the state of the system each time the example is run, but it will look something like:

```
$ python3 -u os_fork_example.py

Child process id: 29190
I am the child
```

After the fork, there are two processes running the same code. For a program to tell which one it is in, it needs to check the return value of `fork()`. If the value is `0`, the current process is the child. If it is not `0`, the program is running in the parent process and the return value is the process id of the child process.

```
# os_kill_example.py

import os
import signal
import time


def signal_usr1(signum, frame):
    "Callback invoked when a signal is received"
    pid = os.getpid()
    print('Received USR1 in process {}'.format(pid))


print('Forking...')
child_pid = os.fork()
if child_pid:
    print('PARENT: Pausing before sending signal...')
    time.sleep(1)
    print('PARENT: Signaling {}'.format(child_pid))
    os.kill(child_pid, signal.SIGUSR1)
else:
    print('CHILD: Setting up signal handler')
    signal.signal(signal.SIGUSR1, signal_usr1)
    print('CHILD: Pausing to wait for signal')
    time.sleep(5)
```

The parent can send signals to the child process using `kill()` and the signal module. First, define a signal handler to be invoked when the signal is received. Then `fork()`, and in the parent pause a short amount of time before sending a USR1 signal using `kill()`. This example uses a short pause to give the child process time to set up the signal handler. A real application, would not need (or want) to call `sleep()`. In the child, set up the signal handler and go to sleep for a while to give the parent time to send the signal.

```
$ python3 -u os_kill_example.py

Forking...
PARENT: Pausing before sending signal...
CHILD: Setting up signal handler
CHILD: Pausing to wait for signal
PARENT: Signaling 29193
Received USR1 in process 29193
```

A simple way to handle separate behavior in the child process is to check the return value of fork() and branch. More complex behavior may call for more code separation than a simple branch. In other cases, there may be an existing program that needs to be wrapped. For both of these situations, the exec*() series of functions can be used to run another program.

```
# os_exec_example.py

import os

child_pid = os.fork()
if child_pid:
    os.waitpid(child_pid, 0)
else:
    os.execlp('pwd', 'pwd', '-P')
```

When a program is run by exec(), the code from that program replaces the code from the existing process.

```
$ python3 os_exec_example.py

.../pymotw-3/source/os
```

There are many variations of exec(), depending on the form in which the arguments are available, whether the path and environment of the parent process should be copied to the child, etc. For all variations, the first argument is a path or filename and the remaining arguments control how that program runs. They are either passed as command line arguments or override the process “environment” (see os.environ and os.getenv). Refer to the library documentation for complete details.

Waiting for Child Processes

Many computationally intensive programs use multiple processes to work around the threading limitations of Python and the global interpreter lock. When starting several processes to run separate tasks, the master will need to wait for one or more of them to finish before starting new ones, to avoid overloading the server. There are a few different ways to do that using wait() and related functions.

When it does not matter which child process might exit first, use wait(). It returns as soon as any child process exits.

```
# os_wait_example.py

import os
import sys
import time

for i in range(2):
    print('PARENT {}: Forking {}'.format(os.getpid(), i))
    worker_pid = os.fork()
    if not worker_pid:
        print('WORKER {}: Starting'.format(i))
        time.sleep(2 + i)
        print('WORKER {}: Finishing'.format(i))
        sys.exit(i)

for i in range(2):
    print('PARENT: Waiting for {}'.format(i))
    done = os.wait()
    print('PARENT: Child done:', done)
```

The return value from wait() is a tuple containing the process id and exit status combined into a 16-bit value. The low byte is the number of the signal that killed the process, and the high byte is the status code returned by the process when it exited.

```
$ python3 -u os_wait_example.py

PARENT 29202: Forking 0
PARENT 29202: Forking 1
PARENT: Waiting for 0
WORKER 0: Starting
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (29203, 0)
PARENT: Waiting for 1
WORKER 1: Finishing
PARENT: Child done: (29204, 256)
```

To wait for a specific process, use `waitpid()`.

```python
# os_waitpid_example.py

import os
import sys
import time

workers = []
for i in range(2):
    print('PARENT {}: Forking {}'.format(os.getpid(), i))
    worker_pid = os.fork()
    if not worker_pid:
        print('WORKER {}: Starting'.format(i))
        time.sleep(2 + i)
        print('WORKER {}: Finishing'.format(i))
        sys.exit(i)
    workers.append(worker_pid)

for pid in workers:
    print('PARENT: Waiting for {}'.format(pid))
    done = os.waitpid(pid, 0)
    print('PARENT: Child done:', done)
```

Pass the process id of the target process, and `waitpid()` blocks until that process exits.

```
$ python3 -u os_waitpid_example.py

PARENT 29211: Forking 0
PARENT 29211: Forking 1
PARENT: Waiting for 29212
WORKER 0: Starting
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (29212, 0)
PARENT: Waiting for 29213
WORKER 1: Finishing
PARENT: Child done: (29213, 256)
```

`wait3()` and `wait4()` work in a similar manner, but return more detailed information about the child process with the pid, exit status, and resource usage.

Spawning New Processes

As a convenience, the `spawn()` family of functions handles the `fork()` and `exec()` in one statement:

```python
# os_spawn_example.py

import os

os.spawnlp(os.P_WAIT, 'pwd', 'pwd', '-P')
```

The first argument is a mode indicating whether or not to wait for the process to finish before returning. This example waits. Use `P_NOWAIT` to let the other process start, but then resume in the current process.

```
$ python3 os_spawn_example.py

.../pymotw-3/source/os
```

Operating System Error Codes

Error codes defined by the operating system and managed in the `errno` module can be translated to message strings using `strerror()`.

```python
# os_strerror.py
```

```python
import errno
import os

for num in [errno.ENOENT, errno.EINTR, errno.EBUSY]:
    name = errno.errorcode[num]
    print('[{num:>2}] {name:<6}: {msg}'.format(
        name=name, num=num, msg=os.strerror(num)))
```

This example shows the messages associated with some error codes that come up frequently.

```
$ python3 os_strerror.py

[ 2] ENOENT: No such file or directory
[ 4] EINTR : Interrupted system call
[16] EBUSY : Resource busy
```

See also

- Standard library documentation for os
- Python 2 to 3 porting notes for os
- `signal` – The section on the `signal` module goes over signal handling techniques in more detail.
- `subprocess` – The `subprocess` module supersedes `os.popen()`.
- `multiprocessing` – The `multiprocessing` module makes working with extra processes easier.
- `tempfile` – The `tempfile` module for working with temporary files.
- Working With Directory Trees – The `shutil` module also includes functions for working with directory trees.
- Speaking UNIX, Part 8. – Learn how UNIX multitasks.
- Standard streams – For more discussion of stdin, stdout, and stderr.
- Delve into Unix Process Creation – Explains the life cycle of a Unix process.
- *Advanced Programming in the UNIX(R) Environment* By W. Richard Stevens and Stephen A. Rago. Published by Addison-Wesley Professional, 2005. ISBN-10: 0201433079 – This book covers working with multiple processes, such as handling signals, closing duplicated file descriptors, etc.

Tracing a Program As It Runs
platform — System Version Information

Quick Links

This page was last updated 2018-03-18.

Navigation

Tracing a Program As It Runs
platform — System Version Information

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?