Fl Studio Scripting Included Python Modules

import gc

The line `import gc` is a Python import statement that includes the `gc` module into your current Python script.

The `gc` module provides an interface to the built-in garbage collector. The garbage collector is responsible for freeing the memory used by objects that are no longer in use by your program.

In this example, we first create a large object that uses a lot of memory. We then delete the object with the `del` statement. However, the memory used by the object might not be freed immediately. To ensure that the memory is freed, we can run the garbage collector manually with `gc.collect()`.

```
# Create a large object that uses a lot of memory
large_object = list(range(1000000))

# Delete the object
del large_object

# Run the garbage collector
gc.collect()
```

import faulthandler

The `faulthandler` module is a built-in Python module that provides functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. It can be useful for post-mortem debugging, especially for programs that might crash unexpectedly.

In this example, the `enable` function is used to enable fault handling. After this line, if your program crashes due to a segmentation fault, a Python traceback will be printed to stderr, which can help you debug the issue.

~~~
Here's a basic example of how you might use the `faulthandler` module:

```python
import faulthandler
faulthandler.enable()

# Rest of your code...
~~~

import errno

The `errno` module defines symbolic names for the error numbers that are set by certain system calls and library functions in the C language. These error numbers are typically used to report errors in system calls and some library functions.
In Python, you can use these symbolic names to understand the type of error encountered during the execution of a system call. For example, if you try to open a file that does not exist, Python raises a `FileNotFoundError` exception. This exception is associated with the `ENOENT` error number, which you can access through the `errno` module.

import builtins

The `builtins` module contains a list of all the standard, built-in functions available in Python like `len()`, `print()`, `type()`, etc. These functions are always available for use in Python code.
However, you don't usually need to import the `builtins` module to use these functions, because they're automatically available in the global namespace. The `builtins` module is typically imported when you want to modify the built-in functions or when you want to check if a function is a built-in function.

```
# Check if a function is a built-in function

if 'print' in dir(builtins):

     print("print is a built-in function.")
```

import binascii

The binascii module provides methods to convert between binary and various ASCII-encoded binary representations. This can be useful when you need to convert binary data to a format that can be included in human-readable contexts (like JSON or XML), or when you need to decode such data.

Here's a basic example of how you might use the binascii module to convert binary data to a hexadecimal string:

```
import binascii

# Some binary data
data = b'\x01\x02\x03\x04\x05'

# Convert the binary data to a hexadecimal string
hex_data = binascii.hexlify(data)

print(hex_data)  # Outputs: b'0102030405'
```

()   Import audioop

~~~
The `audioop` module contains some useful operations on sound fragments. It operates on
sound fragments which are arrays of bytes. It supports a number of basic operations:

- Conversion between different sample types
- Volume adjustment
- Left/right balance
- Adding/multiplying sound fragments
- Auto-gain functions
- More...

Here's a basic example of how you might use the `audioop` module to adjust the volume
of a sound fragment:

```python
import audioop
import wave

# Open a wave file
with wave.open('sound.wav', 'rb') as f:
~~~

```
    # Read the whole file into memory
    fragment = f.readframes(f.getnframes())

# Double the volume
louder_fragment = audioop.mul(fragment, f.getsampwidth(), 2.0)

# Write the result to a new wave file
with wave.open('louder_sound.wav', 'wb') as f:
    f.setparams((f.getnchannels(), f.getsampwidth(), f.getframerate(), f.getnframes(), 
'NONE', 'not compressed'))
    f.writeframes(louder_fragment)
```

In this example, the `mul` function is used to adjust the volume of the sound fragment. The `wave` module is used to read and write wave files.

() Import atexit

The atexit module allows you to register functions that will be called when your program is exiting. This can be useful for cleanup tasks, such as closing files or network connections, or for logging that the program has exited.

Here's a basic example of how you might use the atexit module:

```
import atexit

def goodbye():
    print("Program is exiting...")

# Register the goodbye function to be called at exit
atexit.register(goodbye)
```

() Import _weakref

The `_weakref` module provides mechanisms for creating weak references to objects. A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else.

A primary use for weak references is to implement caches or mappings holding large objects, where it’s desired that a large object not be kept alive solely because it appears in a cache or mapping.

Here's a basic example of how you might use the `_weakref` module:

```python
import _weakref

class MyClass:
    pass

# Create an instance of MyClass
obj = MyClass()

# Create a weak reference to the instance
weak_obj = _weakref.ref(obj)

# The weak reference does not keep the object alive
del obj

# The object has been garbage collected
print(weak_obj())  # Outputs: None
```

In this example, the `ref` function is used to create a weak reference to an instance of `MyClass`. After the original reference to the instance is deleted with `del`, the instance is garbage collected, and the weak reference returns `None`.

() Import _struct

The line `import _struct` is a Python import statement that includes the `_struct` module into your current Python script.

The `_struct` module in Python is typically used for working with C-style binary data formats. It provides pack and unpack functions for working with variable-length binary record formats.

The `pack` function takes a format string and one or more arguments, and returns a bytes object that contains the values packed according to the format string. The `unpack` function does the reverse operation, taking a bytes object and a format string, and returning a tuple of the unpacked values.

Here's a basic example of how you might use the `_struct` module:

```python
import _struct

# Pack some data
packed_data = _struct.pack('hhl', 1, 2, 3)

# Unpack the data
unpacked_data = _struct.unpack('hhl', packed_data)

print(unpacked_data)  # Outputs: (1, 2, 3)
```

In this example, the `pack` function is used to pack the integers 1, 2, and 3 into a bytes object, and the `unpack` function is used to unpack the bytes object back into integers.