:mod:`__main__` --- Top-level code environment

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 4)

Unknown directive type "module".

```
.. module:: __main__
   :synopsis: The environment where top-level code is run. Covers command-line
              interfaces, import-time behavior, and ``__name__ == '__main__'``.
```

In Python, the special name `__main__` is used for two important constructs:

1. the name of the top-level environment of the program, which can be checked using the `__name__ == '__main__'` expression; and
2. the `__main__.py` file in Python packages.

Both of these mechanisms are related to Python modules; how users interact with them and how they interact with each other. They are explained in detail below. If you're new to Python modules, see the tutorial section :ref:`tut-modules` for an introduction.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 16); *backlink*

Unknown interpreted text role "ref".

`__name__ == '__main__'`

When a Python module or package is imported, `__name__` is set to the module's name. Usually, this is the name of the Python file itself without the `.py` extension:

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

If the file is part of a package, `__name__` will also include the parent package's path:

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

However, if the module is executed in the top-level code environment, its `__name__` is set to the string `'__main__'`.

What is the "top-level code environment"?

`__main__` is the name of the environment where top-level code is run. "Top-level code" is the first user-specified Python module that starts running. It's "top-level" because it imports all other modules that the program needs. Sometimes "top-level code" is called an *entry point* to the application.

The top-level code environment can be:

- the scope of an interactive prompt:

  ```
  >>> __name__
  '__main__'
  ```

- the Python module passed to the Python interpreter as a file argument:

  ```
  $ python3 helloworld.py
  Hello, world!
  ```

- the Python module or package passed to the Python interpreter with the :option:`-m` argument:

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 67);

backlink

Unknown interpreted text role "option".

```
$ python3 -m tarfile
usage: tarfile.py [-h] [-v] (...)
```

- Python code read by the Python interpreter from standard input:

```
$ echo "import this" | python3
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

- Python code passed to the Python interpreter with the :option:`-c` argument:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 86); *backlink*

Unknown interpreted text role "option".

```
$ python3 -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

In each of these situations, the top-level module's `__name__` is set to `'__main__'`.

As a result, a module can discover whether or not it is running in the top-level environment by checking its own `__name__`, which allows a common idiom for conditionally executing code when the module is not initialized from an import statement:

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 109)

Unknown directive type "seealso".

```
.. seealso::

   For a more detailed look at how ``__name__`` is set in all situations, see
   the tutorial section :ref:`tut-modules`.
```

Idiomatic Usage

Some modules contain code that is intended for script use only, like parsing command-line arguments or fetching data from standard input. If a module like this was imported from a different module, for example to unit test it, the script code would unintentionally execute as well.

This is where using the `if __name__ == '__main__'` code block comes in handy. Code within this block won't run unless the module is executed in the top-level environment.

Putting as few statements as possible in the block below `if __name__ == '__main__'` can improve code clarity and correctness. Most often, a function named `main` encapsulates the program's primary behavior:

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """A dummy wrapper around print."""
   # for demonstration purposes, you can imagine that there is some
   # valuable and reusable logic inside this function
   print(phrase)

def main() -> int:
```

```
        """Echo the input arguments to standard output"""
        phrase = shlex.join(sys.argv)
        echo(phrase)
        return 0

    if __name__ == '__main__':
        sys.exit(main())  # next section explains the use of sys.exit
```

Note that if the module didn't encapsulate code inside the `main` function but instead put it directly within the `if __name__ == '__main__'` block, the `phrase` variable would be global to the entire module. This is error-prone as other functions within the module could be unintentionally using the global variable instead of a local name. A `main` function solves this problem.

Using a `main` function has the added benefit of the `echo` function itself being isolated and importable elsewhere. When `echo.py` is imported, the `echo` and `main` functions will be defined, but neither of them will be called, because `__name__ != '__main__'`.

Packaging Considerations

`main` functions are often used to create command-line tools by specifying them as entry points for console scripts. When this is done, pip inserts the function call into a template script, where the return value of `main` is passed into :func:`sys.exit`. For example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 167); *backlink*

Unknown interpreted text role "func".

```
sys.exit(main())
```

Since the call to `main` is wrapped in :func:`sys.exit`, the expectation is that your function will return some value acceptable as an input to :func:`sys.exit`; typically, an integer or `None` (which is implicitly returned if your function does not have a return statement).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 175); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 175); *backlink*

Unknown interpreted text role "func".

By proactively following this convention ourselves, our module will have the same behavior when run directly (i.e. `python3 echo.py`) as it will have if we later package it as a console script entry-point in a pip-installable package.

In particular, be careful about returning strings from your `main` function. :func:`sys.exit` will interpret a string argument as a failure message, so your program will have an exit code of `1`, indicating failure, and the string will be written to :data:`sys.stderr`. The `echo.py` example from earlier exemplifies using the `sys.exit(main())` convention.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 185); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 185); *backlink*

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 191)

Unknown directive type "seealso".

```
.. seealso::

   `Python Packaging User Guide <https://packaging.python.org/>`_
   contains a collection of tutorials and references on how to distribute and
   install Python packages with modern tools.
```

`__main__.py` in Python Packages

If you are not familiar with Python packages, see section :ref:`tut-packages` of the tutorial. Most commonly, the `__main__.py` file is used to provide a command-line interface for a package. Consider the following hypothetical package, "bandclass":

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 201); *backlink***
>
> Unknown interpreted text role "ref".

```
bandclass
  â”œâ”€â”€ __init__.py
  â”œâ”€â”€ __main__.py
  â””â”€â”€ student.py
```

`__main__.py` will be executed when the package itself is invoked directly from the command line using the :option:`-m` flag. For example:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 213); *backlink***
>
> Unknown interpreted text role "option".

```
$ python3 -m bandclass
```

This command will cause `__main__.py` to run. How you utilize this mechanism will depend on the nature of the package you are writing, but in this hypothetical case, it might make sense to allow the teacher to search for students:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[2] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Note that `from .student import search_students` is an example of a relative import. This import style can be used when referencing modules within a package. For more details, see :ref:`intra-package-references` in the :ref:`tut-modules` section of the tutorial.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 233); *backlink***
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 233); *backlink***
>
> Unknown interpreted text role "ref".

Idiomatic Usage

The contents of `__main__.py` typically isn't fenced with `if __name__ == '__main__'` blocks. Instead, those files are kept short, functions to execute from other modules. Those other modules can then be easily unit-tested and are properly reusable.

If used, an `if __name__ == '__main__'` block will still work as expected for a `__main__.py` file within a package, because its `__name__` attribute will include the package's path if imported:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won't work for `__main__.py` files in the root directory of a .zip file though. Hence, for consistency, minimal `__main__.py` like the :mod:`venv` one mentioned below are preferred.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 254); *backlink***
>
> Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 258)

Unknown directive type "seealso".

```
.. seealso::

   See :mod:`venv` for an example of a package with a minimal ``__main__.py``
   in the standard library. It doesn't contain a ``if __name__ == '__main__'``
   block. You can invoke it with ``python3 -m venv [directory]``.

   See :mod:`runpy` for more details on the :option:`-m` flag to the
   interpreter executable.

   See :mod:`zipapp` for how to run applications packaged as *.zip* files. In
   this case Python looks for a ``__main__.py`` file in the root directory of
   the archive.
```

import __main__

Regardless of which module a Python program was started with, other modules running within that same program can import the top-level environment's scope (:term:`namespace`) by importing the `__main__` module. This doesn't import a `__main__.py` file but rather whichever module that received the special name `'__main__'`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 276); *backlink*

Unknown interpreted text role "term".

Here is an example module that consumes the `__main__` namespace:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    if '__file__' in dir(__main__):
        print(__main__.my_name, "found in file", __main__.__file__)
    else:
        print(__main__.my_name)
```

Example usage of this module could be as follows:

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"

def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Now, if we started our program, the result would look like this:

```
$ python3 start.py
Define the variable `my_name`!
```

The exit code of the program would be 1, indicating an error. Uncommenting the line with `my_name = "Dinsdale"` fixes the program and now it exits with status code 0, indicating success:

```
$ python3 start.py
Dinsdale found in file /path/to/start.py
```

Note that importing `__main__` doesn't cause any issues with unintentionally running top-level code meant for script use which is put in the `if __name__ == "__main__"` block of the `start` module. Why does this work?

Python inserts an empty `__main__` module in :attr:`sys.modules` at interpreter startup, and populates it by running top-level code. In our example this is the `start` module which runs line by line and imports `namely`. In turn, `namely` imports `__main__` (which is really `start`). That's an import cycle! Fortunately, since the partially populated `__main__` module is present in :attr:`sys.modules`, Python passes that to `namely`. See :ref:`Special considerations for __main__ <import-dunder-main>` in the import system's reference for details on how this works.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 339); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 339); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 339); *backlink*

Unknown interpreted text role "ref".

The Python REPL is another example of a "top-level environment", so anything defined in the REPL becomes part of the `__main__` scope:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

Note that in this case the `__main__` scope doesn't contain a `__file__` attribute as it's interactive.

The `__main__` scope is used in the implementation of :mod:`pdb` and :mod:`rlcompleter`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 367); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)__main__.rst, line 367); *backlink*

Unknown interpreted text role "mod".