HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW09

Modules and Packages

Information Technology

April 17, 2025

FH Zentralschweiz

Content

- Modules
  - Concept (relative and absolute path)
  - Import (all *, specific attributes (from), as, import)
- If __name__ == “__main__”
- Packages
  - Difference to Folder Structure
  - The __init__.py script
  - Deploy a script or package
- External Packages
  - What is PIP
  - Install Packages
  - Example on pandas and matplotlib
- Exercises

HSLU Lucerne University of Applied Sciences and Arts

Modules

Information Technology

April 17, 2025

FH Zentralschweiz

Modules: Concept

“A module is a file containing Python definitions and statements.”

We can use modules for better structuring the source code of the whole application.

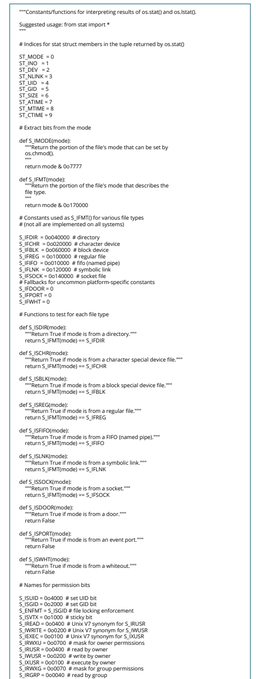
This allows for:

1. having code more organized and easier maintenance.
2. developing libraries that can be used in different applications.

A module can contain:

- Definitions like variables or Enums
- Functions
- Classes
- Structs
- e.t.c.

`/usr/lib/python3.12/stat.py`



```
"""Constants/functions for interpreting results of os.stat() and os.lstat().

Suggested usage: from stat import *
"""

# Indices for stat struct members in the tuple returned by os.stat()

ST_MODE  = 0
ST_INO   = 1
ST_DEV   = 2
ST_NLINK = 3
ST_UID   = 4
ST_GID   = 5
ST_SIZE  = 6
ST_ATIME = 7
ST_MTIME = 8
ST_CTIME = 9

# Extract bits from the mode

def S_IMODE(mode):
    """Return the portion of the file's mode that can be set by
    os.chmod().
    """
    return mode & 0o7777

def S_IFMT(mode):
    """Return the portion of the file's mode that describes the
    file type.
    """
    return mode & 0o170000

# Constants used as S_IFMT() for various file types
# (not all are implemented on all systems)

S_IFDIR  = 0o040000  # directory
S_IFCHR  = 0o020000  # character device
S_IFBLK  = 0o060000  # block device
S_IFREG  = 0o100000  # regular file
S_IFIFO  = 0o010000  # fifo (named pipe)
S_IFLNK  = 0o120000  # symbolic link
S_IFSOCK = 0o140000  # socket file
# Fallbacks for uncommon platform-specific constants
S_IFDOOR = 0
S_IFPORT = 0
S_IFWHT = 0

# Functions to test for each file type

def S_ISDIR(mode):
    """Return True if mode is from a directory."""
    return S_IFMT(mode) == S_IFDIR

def S_ISCHR(mode):
    """Return True if mode is from a character special device file."""
    return S_IFMT(mode) == S_IFCHR

def S_ISBLK(mode):
    """Return True if mode is from a block special device file."""
    return S_IFMT(mode) == S_IFBLK

def S_ISREG(mode):
    """Return True if mode is from a regular file."""
    return S_IFMT(mode) == S_IFREG

def S_ISFIFO(mode):
    """Return True if mode is from a FIFO (named pipe)."""
    return S_IFMT(mode) == S_IFIFO

def S_ISLNK(mode):
    """Return True if mode is from a symbolic link."""
    return S_IFMT(mode) == S_IFLNK

def S_ISSOCK(mode):
    """Return True if mode is from a socket."""
    return S_IFMT(mode) == S_IFSOCK

def S_ISDOOR(mode):
    """Return True if mode is from a door."""
    return False

def S_ISPORT(mode):
    """Return True if mode is from an event port."""
    return False

def S_ISWHT(mode):
    """Return True if mode is from a whiteout."""
    return False

# Names for permission bits

S_ISUID = 0o4000  # set UID bit
S_ISGID = 0o2000  # set GID bit
S_ENFMT = S_ISGID # file locking enforcement
S_ISVTX = 0o1000  # sticky bit
S_IREAD = 0o0400  # Unix V7 synonym for S_IRUSR
S_IWRITE = 0o0200 # Unix V7 synonym for S_IWUSR
S_IEXEC = 0o0100  # Unix V7 synonym for S_IXUSR
S_IRWXU = 0o0700  # mask for owner permissions
S_IRUSR = 0o0400  # read by owner
S_IWUSR = 0o0200  # write by owner
S_IXUSR = 0o0100  # execute by owner
S_IRWXG = 0o0070  # mask for group permissions
S_IRGRP = 0o0040  # read by group
```

Modules: Import

In order to access content of other modules, they can be **imported** to a module as **whole** or **parts** of it.

- The name of a module is the file name **without** the file extension (`.py`).
- With reference to the module name, we can import selected parts of it.

Assume we have the given module `hi.py`. The module (or parts of it) can be imported by:

`import hi`	declares reference to module `hi.py`
`import hi as hi_mod`	renames reference to `hi_mod`
`from hi import Welcome`	declares reference to specific attribute
`from hi import *`	declares references to “all” attributes

`hi.py`

```
hi_term = 'hello world'

def say_hi(name=hi_term):
    print(name)

class Welcome:
    def __init__(self, name):
        self.name = name

    def say_welcome(self):
        say_hi(self.name)
```

Modules: Import

When we **refer** to a module for importing all or any parts of it, the **full** module is interpreted.

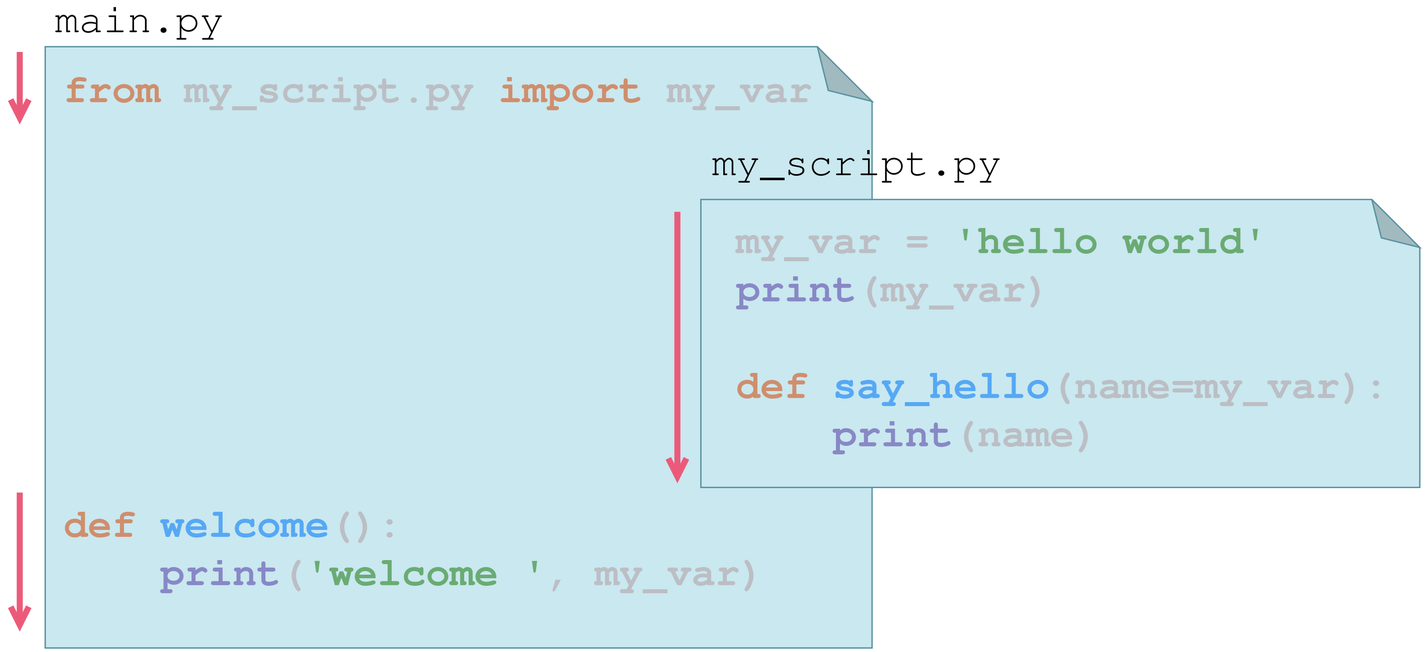
This means, when **referring** to a module with the `import`, the interpreter starts at the top of the referred module and goes through **all** the content. Thereby, all statements and declarations are executed.

This holds for the import of a whole module:

```
import path.to.module
```

as well as for parts of it:

```
from path.to.module import attribute
```



Modules: Import

When imported, attributes of external modules are accessed differently, depending on the import.

Assume we have the given module `hi.py`. The module (or parts of it) can be accessed by:

Import	Access
`import hi`	`my_var = hi.hi_term`
`import hi as hi_mod`	`my_var = hi_mod.hi_term`
`from hi import Welcome`	`my_var = Welcome('Peter')`
`from hi import *`	`my_var = say_hi()`

hi.py

```
hi_term = 'hello world'

def say_hi(name=hi_term):
    print(name)

class Welcome:
    def __init__(self, name):
        self.name = name

    def say_welcome(self):
        say_hi(self.name)
```

Modules: Import

Referring to external modules and files require either absolute or relative references.

- By default, root is set to the main executed script.
- The **main** executed script only allows **absolute** path.
- Docu: https://docs.python.org/3/tutorial/modules.html#importing-from-a-package

```
a/
 | a.py
 | aa.py
 |
 |-b/
    | b.py
    |
    |-c/
    |  | c.py
    |
    |-cc/
       | cc.py
```

Absolute path: (assume: `python b.py; import in b.py`)

`import c.c`	Full module `c.py`
`from cc.cc import var`	Import `var` of module `cc.py`
~~`import a.b.c`~~	Import beyond top-level

Relative path – no direct import!: (assume: `python a.py`)

`from .c.c import var`	In `b.py`: import `var` of module `c.py`
~~`from ..aa import var`~~	In `b.py`: beyond top-level; abs required
`from ..cc.cc import var`	In `c.py`: import `var` of module `cc.py`

Modules: Import

By default, imported modules are searched relative to the script path. Yet, some times scripts are in a different location (e.g. collection of personal libraries) where the interpreter also should search for imported scripts.

For this case, the module **`sys`** allows to extend the search space for imported modules by:

```
sys.path.append('C:\\Users\\additional\\location')          Windows
sys.path.append('/home/username/additional/location')       Linux
```

note: each path must be **absolute!**

Modules: Import

Python allows some tweaks for importing attributes from modules and packages. These include multiple imports on a single line as well as the asterisk operator:

Assuming project structure on previous slide, we can import multiple attributes by:

- `from b.b import var1, var2` or
- `from b.b import var1 as v1, var2 as v2`

Alternatively, the asterisk operator (*****) allows to import all attributes by name. For the example above, this means that the `var1` and `var2` are accessible by name **without** module reference.

- `from b.b import *`

However, for large modules referred to, this **blows up** the final sources unnecessarily.

Both, chained imports as well as the asterisk operator should not be used as by convention:

https://peps.python.org/pep-0008/

HSLU Lucerne University of Applied Sciences and Arts

If __name__ == “__main__”

Information Technology

April 17, 2025

FH Zentralschweiz

If __name__ == “__main__”

When the interpreter declares a module (interpreting all content of a module), it assigns a unique name to it.

- The module name assigned to the interpreted module equals the path from root (dot separated) + the file name without the appendix .py.
- The module name of `my_mod.py` is:
  `path.to.module.my_mod`

The module name of the **first** interpreted module (i.e. the main module) is set to **__main__**.

Consequently, we can execute some tests in an if clause that are only executed if the module is the **main module**.

```
from s.th import attr

some_var = 'hello world'

if __name__ == '__main__':
    print(f"{some_var=}")
```

HSLU Lucerne University of Applied Sciences and Arts

Packaging

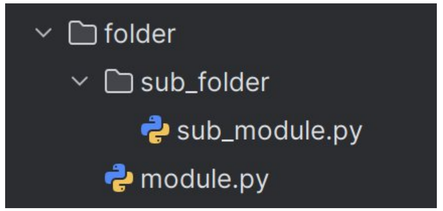
Information Technology

April 17, 2025

Packaging: Difference to Folder Structure

Python considers a regular system **folder** as a simple **structural element** that can contain:

- modules (.py) or resource files such as
- data files (.json, .csv, ...),
- Text (.txt),
- Figures (.jpg, .png, ...) or
- Icons (.ico)



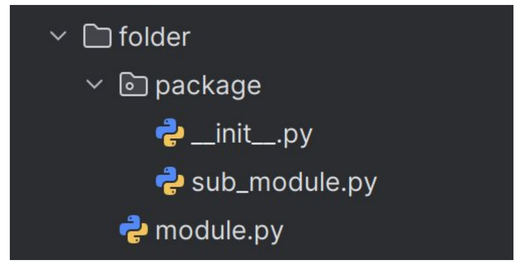
Python can import modules (.py) and objects (global variables, functions, classes, ...) from folders using the **dot-notation**.

```
from path.to.folder import module
```

Packaging: Difference to Folder Structure

In addition, Python enables to structure project resources using **packages**.

- a folder becomes a package as soon as it comprises
  a script named: **`__init__.py`**



A package, from structural point of view, is equally to folders comprising various project resources. Consequently, content is imported in similar ways.

```
from path.to.package import module
```

Contrasting to folders, a **package** can also be **imported** similar to regular modules. The package, namely the script **`__init__.py`** is also invoked when a module from the package, any object of it or the package itself is imported. (except: namespace packages)

Packaging: The `__init__.py` Script

Like a module that gets imported, the `__init__.py` script will be interpreted when a package or parts of it get imported.

The script `__init__.py` can:

- be empty,
- import additional modules or
- define variables, functions, classes, etc.
- be used to keep an interface of a module when it is turned into a package

Objects defined in the `__init__.py` are bound to names in the **package's namespace**.

- The __name__ of __init__.py is the package name.

Doc: https://docs.python.org/3/reference/import.html#regular-packages

Packaging: Deployment

As we all know, Python is a script language.

- Distributing packages does not require any special compilation nor packing.

Sharing library functions or modules with colleagues or project contributors, can easily be done with exchanging Python modules, packages or directories.

Following some additional deployment constraints, packages can also be provided to the official python package repository PyPI.

Docs: https://packaging.python.org/en/latest/tutorials/packaging-projects/

```
/usr/lib/python3.12/site-packages/tbb:
total 136K
-rwxr-xr-x 1 root  66K Nov 13 23:54 _api.cpython-312-x86_64-linux-gnu.so
-rw-r--r-- 1 root 5.2K Nov 13 23:54 api.py
-rw-r--r-- 1 root  13K Nov 13 23:54 __init__.py
-rw-r--r-- 1 root  647 Nov 13 23:54 __main__.py
-rw-r--r-- 1 root  25K Nov 13 23:54 pool.py
drwxr-xr-x 2 root 4.0K Nov 25 20:56 __pycache__
-rw-r--r-- 1 root 7.0K Nov 13 23:54 test.py

/usr/lib/python3.12/site-packages/tbb/__pycache__:
total 72K
-rw-r--r-- 1 root 7.8K Nov 13 23:54 api.cpython-312.pyc
-rw-r--r-- 1 root  17K Nov 13 23:54 __init__.cpython-312.pyc
-rw-r--r-- 1 root  254 Nov 13 23:54 __main__.cpython-312.pyc
-rw-r--r-- 1 root  27K Nov 13 23:54 pool.cpython-312.pyc
-rw-r--r-- 1 root 8.1K Nov 13 23:54 test.cpython-312.pyc

/usr/lib/python3.12/site-packages/TBB-0.2-py3.12.egg-info:
total 16K
-rw-r--r-- 1 root    1 Nov 13 23:54 dependency_links.txt
-rw-r--r-- 1 root 1.3K Nov 13 23:54 PKG-INFO
-rw-r--r-- 1 root  210 Nov 13 23:54 SOURCES.txt
```

External Packages

External Packages: What is pip?

The Python Software Foundation hosts an own repository for ‘official’ Python packages called: PyPI (pypi.org)

pip is a command-line interface (CLI) tool for installing Python packages.

- It is the one included with modern versions of Python.

Most frequently used pip commands are:

- `install` Install packages.
- `download` Download packages.
- `uninstall` Uninstall packages.
- `freeze` Output installed packages in requirements format.
- `list` List installed packages.
- `show` Show information about installed packages.
- `help` Show help for commands.

External Packages: Install Packages

Packages can be installed with pip from different sources including:

- .zip packages
- version control systems (Git, Bitbucket, etc. ..)
- pypi

For installing a certain package, from the official pypi repository, follow the steps:

1. Look for desired package on pypi.org
2. Open CLI and make sure pip is installed
3. Run: pip install <package_name>

Simply change the URL for different repositories:

- pip install git+https://github.com/user/repo.git@develop

External Packages: Example on Pandas and Matplotlib

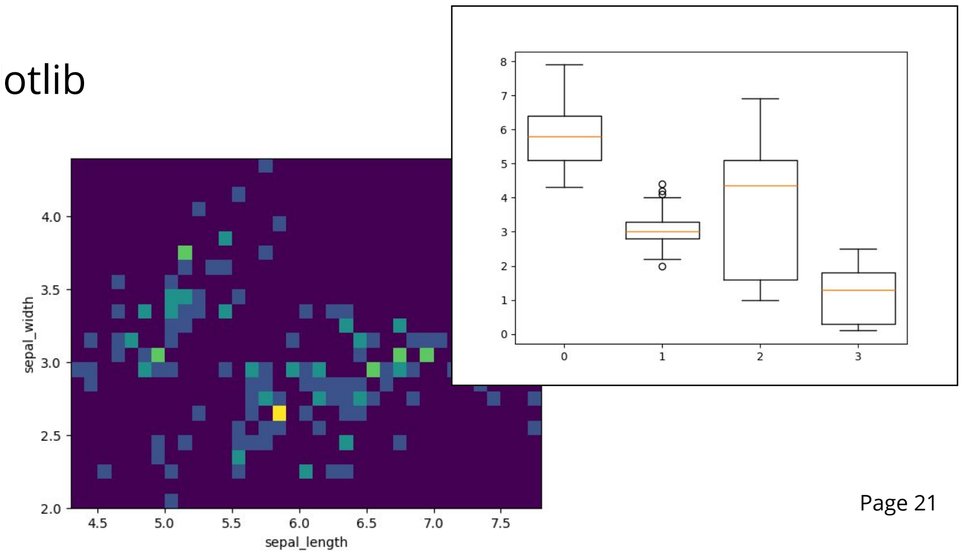
As an exercise to get familiar with Python libraries, install the two often used libraries for data science:

- Pandas
- Matplotlib

1. Download and import the iris data set to a global variable from:
   - Second source: https://gist.github.com/curran/a08a1080b88344b0c8a7
   - Note, for ease use the raw data link: https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/0e7a9b0a5d22642a06d3d5b9bcbad9890c8ee534/iris.csv

2. Create a new Python script file and import the libraries: pandas and matplotlib
   - `import pandas as pd`
   - `import matplotlib as mp`

3. Inspect the downloaded data set with first plots using matplotlib
   Doc: https://matplotlib.org/stable/plot_types/index.html



HSLU Lucerne University of Applied Sciences and Arts

Exercises

Information Technology

April 17, 2025

Exercises

On the **vm** we provided an exercise to modules and import. This exercise comprises the files:

- Run_script.py (main script)
- String_operations.py (additional module)

HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch