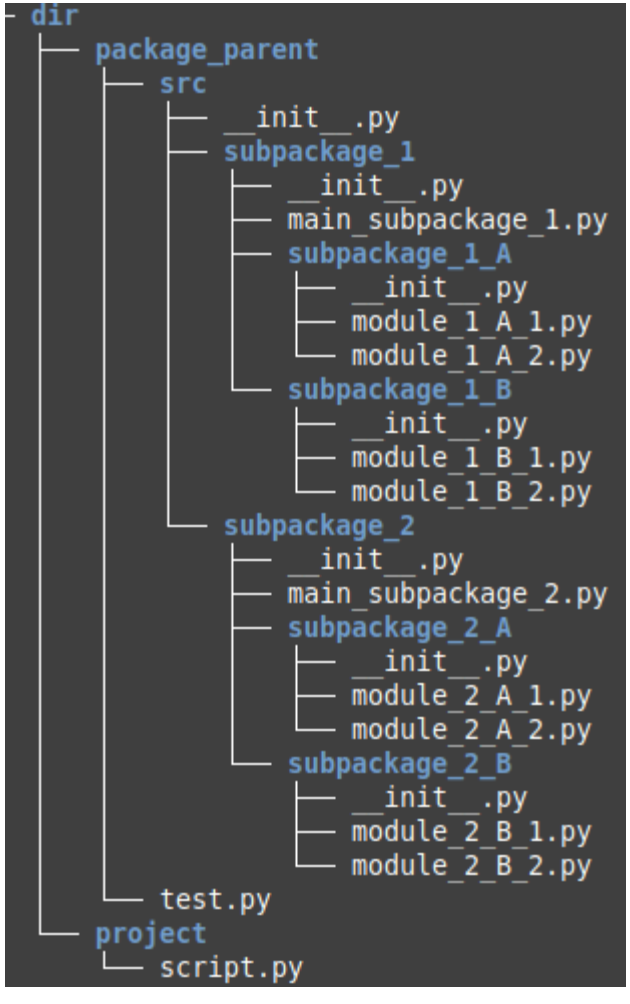
Introduction

To import some content, Python must know:

- which directory is a package
- where to find such directories

Hereafter is presented a simple methodology to access some Python code that is stored elsewhere on the disk.

Example



There are two directories sharing the same parent:

- `package_parent` : hosts the `src` directory which contains a Python package.

  In `src` , the first `__init__.py` file tells Python that directory `src` is a **package**. The other `__init__.py` files define a subpackages whose name are the directory they are in (`subpackage_1` , `subpackage_1_A` , etc...). The other `*.py` files are those containing useful code. They are called **modules**. Each of them is filled with:

  - a `print` that tells about the file name
  - a variable `var`

- `project` : a fictive Python project that contains a Python file

Search for packages

When importing a package, Python looks for it:

- in the current directory
- in files described in the `sys.path` list

Thus, the import of `src` from the `project` directory (for instance in `script.py`) will fail as current directory (`project`) is not the one of the package (`package_parent`).

In [1]:
```
import os
os.chdir(r'dir/project')
import src
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import os
      2 os.chdir(r'dir/project')
----> 3 import src

ModuleNotFoundError: No module named 'src'
```

Let's move to `package_parent`:

In [2]:
```
os.chdir(r'../package_parent/')
import src
```

```
I am __init__ of package
```

Import succeeded.

Yet, changing current directory is not a good solution to access packages. **The preferred way is to modify the `sys.path` variable**. The path is inserted at first position:

In [3]:
```
os.chdir(r'../project')    # back to a dir different than the one compatbile with imp
from sys import path
from pathlib import Path
path.insert(0, str(Path('../package_parent/').resolve()))
```

Then the import is running smoothly:

In [4]:
```
%reset -f --aggressive
import src
```

```
culling sys module...
I am __init__ of package
```

Note: in Python console, packages/modules are never imported twice. Thus, for explanation purpose, the magic function `reset -f` is used here to erase all the memory content, included imports, so that we can experience a second import. Usually, this is not necessary (and `--agressive` must not be used).

Add some content to __init__

When importing a module, one can access its components:

In [5]:
```
%reset -f --aggressive
from src.subpackage_1.subpackage_1_A import module_1_A_1
module_1_A_1.var
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_A
I am module_1_A_1
```

Out[5]:
```
1
```

Conversely, when importing a package 'my_package', with an empty `__init__.py` file, modules or subpackages that are contained in 'my_package' are not imported. In the code snippet below, access to a module of 'package' ('module_1_B_1') is impossible:

In [6]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_B as package
package.module_1_B_1.var
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_B
I am __init__ of subpackage_1_A
I am module_1_A_1
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 3
      1 get_ipython().run_line_magic('reset', '-f --aggressive')
      2 import src.subpackage_1.subpackage_1_B as package
----> 3 package.module_1_B_1.var

AttributeError: module 'src.subpackage_1.subpackage_1_B' has no attribute 'm
odule_1_B_1'
```

Suppose we add some additionnal content to `module_1_A_1`:

```
print("I am module_1_A_1")
var = 1

def new_func():
    print("I am `new_func`")
```

To import this content when importing `subpackage_1_A`, the file `src/subpackage_1/subpackage_1_A/__init__.py` is completed with:

```
print("I am __init__ of subpackage_1_A")
useless_var = 10
from .module_1_A_1 import var
from .module_1_A_1 import new_func as imported_func
```

The syntax `.[module_name]` tells Python to search for a module in the current package.

Then, back to our project, the import of module content does not fail:

In [7]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_A as package
print(package.useless_var)
print(package.var)
package.imported_func()
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_A
I am module_1_A_1
10
1
I am `new_func`
```

Explanation: `import src.subpackage_1.subpackage_1_A as package` executed the instructions contained in the `__init__` file of `subpackage_1_A`. These instructions make avilable some content of `module_1_A_1`.

The search is possible in siblings packages or parent packages using the `..` syntax. For instance, let's import the same components but from `module_1_B_1.py` by modifying `src/subpackage_1/subpackage_1_B/__init__.py`:

```
print("I am __init__ of subpackage_1_B")
from ...subpackage_1.subpackage_1_A import imported_func
```

In [8]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_B as package
package.imported_func()
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_B
I am __init__ of subpackage_1_A
I am module_1_A_1
I am `new_func`
```

Note: one can also use the `from module import *` syntax to import everything from a module:

```
print("I am __init__ of subpackage_1_B")
from ...subpackage_1.subpackage_1_A import *
```

Yet, this syntax must be avoided.

Take away

Never copy/paste package directories to make import easier!

Packages and modules

- a module is a `*.py` file within a package
- a package is a directory containing an `__init__.py` file
- file `__init__.py` is ran when the package is imported
- some content to be imported can be added to `__init__.py` files using the `.`, `..`, `...` (etc...) syntax to browse sibling subpackages
- a content is never imported twice

`sys.path`

- Modifying `sys.path` is a way to access some Python code stored elsewhere on the disk.
- One must **never** copy/paste package directories to make import easier.