__init__.py

PYTHON: PROGRAMMING GUIDELINES FOR MODULES

ALEXANDRE BOUCAUD
DEVELOPERS WORKSHOP #2 - GENEVA, OCT 15

OUTLINE

- What is a Python module/package/project ?
- The role of the `__init__.py`
- The rules for maintaining big Python projects in the EDEN framework

A PYTHON MODULE

https://docs.python.org/2/tutorial/modules.html

- a module / package / project often refer to the same thing
- a single file or a set of files containing definitions
- organized in a single directory or several directories (submodules)
- can be imported into an other module or into an interactive Python kernel

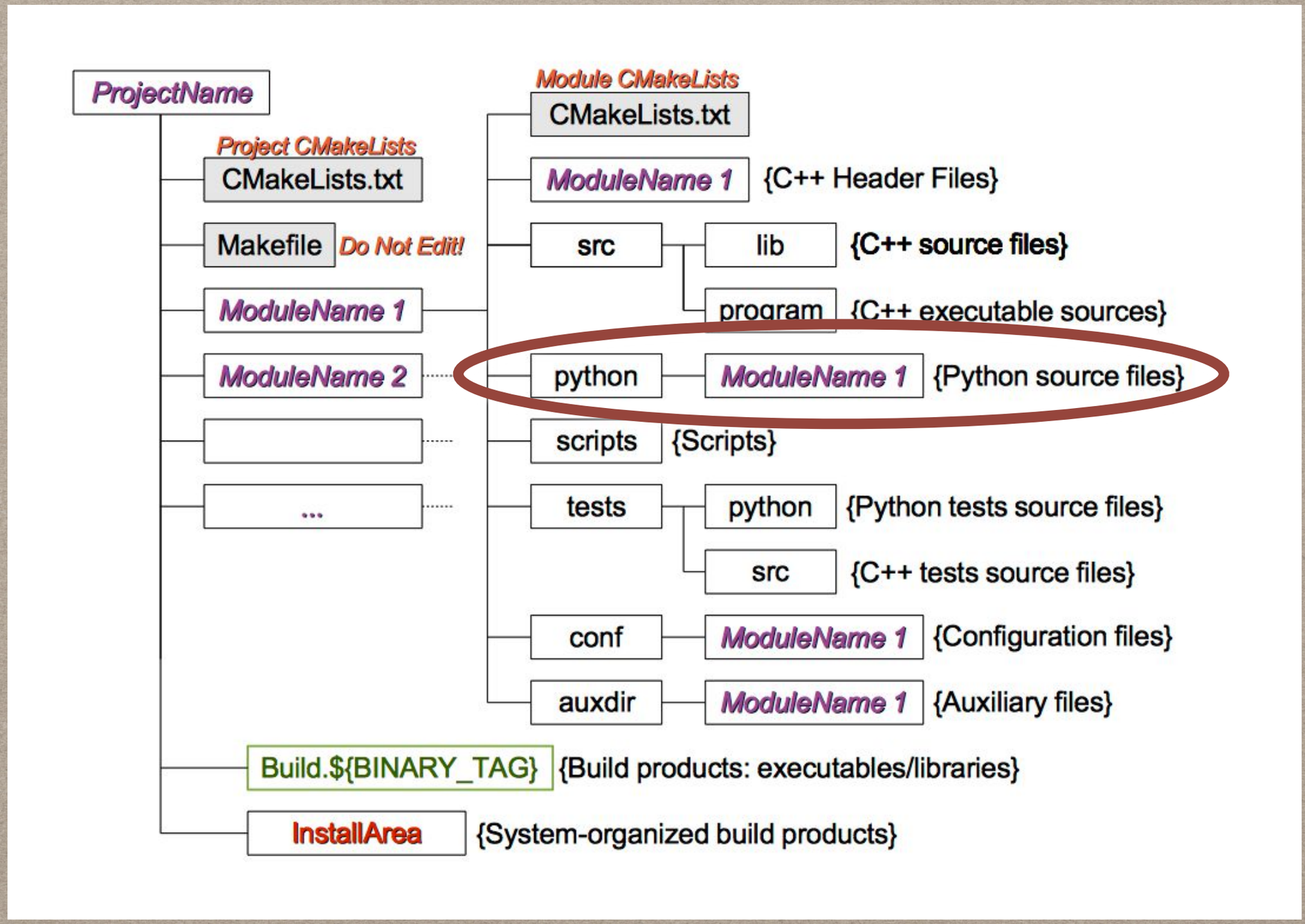
Simplest project

```
— myfirstmodule.py
```

Euclid-like project

```
└── modulename
    ├── __init__.py
    ├── main_defs.py
    ├── module_classes.py
    ├── submodule_a
    |   ├── __init__.py
    |   ├── some_funcs.py
    |   └── some_more_funcs.py
    └── submodule_b
        ├── __init__.py
        ├── some_defs.py
        └── some_more_defs.py
```

WHERE IS THE PYTHON MODULE IN AN ELEMENTS PROJECT DIRECTORY



IMPORTING A MODULE

```python
# General imports

import modulename

import modulename.main_defs

import modulename.submodule_a.some_funcs

# Specific imports

from modulename.some_more_funcs import func1, func2

from modulename.submodule_b import some_defs

# Redefining the module name

import modulename.module_classes as myclasses

from modulename.submodule_a import some_funcs as myfuncs
```

example project structure

```
└── modulename
    ├── __init__.py
    ├── main_defs.py
    ├── module_classes.py
    ├── submodule_a
    |   ├── __init__.py
    |   ├── some_funcs.py
    |   └── some_more_funcs.py
    └── submodule_b
        ├── __init__.py
        ├── some_defs.py
        └── some_more_defs.py
```

DIR()

- `dir()` is a built-in function
- `dir()` **finds the names that a module defines on import**
- names can be built-in, global and local variables, methods, classes as well as submodules
- corresponds to the names available on autocompletion after « . »
- it is **used to check interactively if imports worked as desired**

Example

```
In [1]: import time

In [2]: dir(time)
Out[2]:
['__doc__',
 '__file__',
 '__name__',
 '__package__',
 'accept2dyear',
 'altzone',
 'asctime',
 'clock',
 'ctime',
 'daylight',
 'gmtime',
 'localtime',
 'mktime',
 'sleep',
 'strftime',
 'strptime',
 'struct_time',
 'time',
 'timezone',
 'tzname',
 'tzset']

In [3]: time.|   + hit tab
```

ROLE OF THE __INIT__.PY

- **organize** your project into several directories (submodules)

- **clarify** which and how methods/classes will be imported

- **document** the (sub-)module itself

ORGANIZE CODE

- write **human-sized** source code files
- gather methods with common topic
- no requirement on the number of methods/ classes in a single file BUT **< 10** is a good guess

Example

```
├── messy_project
|   ├── __init__.py
|   └── all_the_methods.py
```

vs.

```
└── organized_project
    ├── __init__.py
    ├── project_classes.py
    └── utils
        ├── __init__.py
        ├── algorithms.py
        └── fits.py
```

ORGANIZE CODE

- write **human-sized** source code files
- gather methods with common topic
- no requirement on the number of methods/ classes in a single file BUT **< 10** is a good guess
- be careful on cross-imports within a module

Example

```
├── messy_project
|   ├── __init__.py
|   └── all_the_methods.py
```

vs.

```
└── organized_project
    ├── __init__.py
    ├── project_classes.py
    └── utils
        ├── __init__.py
        ├── algorithms.py
        └── fits.py
```

file_a.py

```
from file_b import foo
```

file_b.py

```
from file_a import bar
```

ORGANIZE CODE

- write **human-sized** source code files
- gather methods with common topic
- no requirement on the number of methods/ classes in a single file BUT **< 10** is a good guess
- be careful on cross-imports within a module

Example

```
├── messy_project
|   ├── __init__.py
|   └── all_the_methods.py
```

vs.

```
└── organized_project
    ├── __init__.py
    ├── project_classes.py
    └── utils
        ├── __init__.py
        ├── algorithms.py
        └── fits.py
```

file_a.py

```
from file_b import foo
```

file_b.py

```
from file_a import bar
```

CLARIFY THE MODULE IMPORTS

- the syntax `from package import *` should only be written in `__init__.py` files

- « * » means « all » and refers to the Python `__all__` variable

- this variable is a list of methods (functions or classes) or variables that will be imported with an `import *`

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)

```

__init__.py

```python
from .module import *

```

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

__init__.py

```python
from . import module
```

console

__init__.py

```python
from .module import *
```

console

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

__init__.py

```python
from .module import *
```

console

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

__init__.py

```python
from .module import *
```

console

importing the top module does not import submodules

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module']

In [3]: project.module.my_func1()
Out[3]: 'Hello world!'
```

__init__.py

```python
from .module import *
```

console

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module']

In [3]: project.module.my_func1()
Out[3]: 'Hello world!'
```

__init__.py

```python
from .module import *
```

console

importing the submodule in the __init__.py changes that

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module']

In [3]: project.module.my_func1()
Out[3]: 'Hello world!'
```

__init__.py

```python
from .module import *
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module',
 'my_func1']

In [3]: project.my_func1()
Out[3]: 'Hello world!'
```

CLARIFY THE MODULE IMPORTS

Example

```
└── project
    ├── __init__.py
    └── module.py
```

module.py

```python
from math import sqrt, cos

__all__ = ['my_func1']

def my_func1():
    return "Hello world!"

def my_func2(arg):
    return sqrt(arg)
```

__init__.py

empty

console

```
In [1]: import project

In [2]: dir(project)
Out[2]: ['__builtins__', '__doc__', '
__file__', '__name__', '__package__',
 '__path__']

In [3]: import project.module

In [4]: project.module.my_func1()
Out[4]: 'Hello world!'
```

__init__.py

```python
from . import module
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module']

In [3]: project.module.my_func1()
Out[3]: 'Hello world!'
```

__init__.py

```python
from .module import *
```

console

```
In [1]: import project

In [2]: dir(project)
Out[2]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'module',
 'my_func1']

In [3]: project.my_func1()
Out[3]: 'Hello world!'
```

importing * from the submodule also brings the __all__ attributes

DOCUMENT THE MODULE

- documenting methods is capital *see Marco's talk this afternoon*
- **documenting modules is** equally as **important** if you want your code to be explored interactively
- to this extent, the `__init__.py` **should have a docstring**

Example

```
└── project
    ├── __init__.py
    └── mainmodule.py
```

__init__.py

```
"""
This is the project documentation
"""
from .mainmodule import *
```

in your Python kernel

```
In [1]: import project

In [2]: project?
Type:        module
String form: <module 'project' from 'project/__init__.pyc'>
File:
~/work/Euclid/devel/DevWorkshop2/doc_variable/project/__init__.py
Docstring:   This is the project documentation
```

DOCUMENT THE MODULE

- documenting methods is capital *see Marco's talk this afternoon*
- **documenting modules is** equally as **important** if you want your code to be explored interactively
- to this extent, the `__init__.py` **should have a docstring**

Example

```
└── project
    ├── __init__.py
    └── mainmodule.py
```

__init__.py

```
"""
This is the project documentation
"""
from .mainmodule import *
```

in your Python kernel

```
In [1]: import project

In [2]: project?
Type:        module
String form: <module 'project' from 'project/__init__.pyc'>
File:
~/work/Euclid/devel/DevWorkshop2/doc_variable/project/__init__.py
Docstring:   This is the project documentation
```

A FEW EXAMPLES TO AVOID IN THE __INIT__.PY

*absolute imports with import **

```python
"""
Please do not use absolute imports
in the __init__.py
"""
from module.utils.fits import *
from module.utils.algorithms import *
```

imports from other packages

```python
"""
Please do not put other imports than
the module ones in the __init__.py
"""
import numpy as np

from .fits import *
from .algorithms import *
```

new definitions

```python
"""
Please do not define methods
or classes inside the __init__.py
"""
from .fits import *
from .algorithms import *

def helloworld():
    print "Hello world !"
```

A (QUICK) TOUR OF EXISTING GUIDELINES ON THIS

THROUGH BIG ASTRO-PROJECTS (ASTROPY AND LSST)

ASTROPY

http://docs.astropy.org/en/stable/development/codeguide.html

- if a module is small enough it should be a single file

- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.

ASTROPY

http://docs.astropy.org/en/stable/development/codeguide.html

- if a module is small enough it should be a single file
- relative imports allowed within module to improve modularity

- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.

- One exception is to be made from the PEP8 style: new style relative imports of the form `from . import modname` are allowed and required for Astropy, as opposed to absolute (as PEP8 suggests) or the simpler `import modname` syntax. This is primarily due to improved relative import support since PEP8 was developed, and to simplify the process of moving modules.

ASTROPY

http://docs.astropy.org/en/stable/development/codeguide.html

- if a module is small enough it should be a single file
- relative imports allowed within module to improve modularity
- general utility methods should be put in a separate module for clarity

- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.

- One exception is to be made from the PEP8 style: new style relative imports of the form `from . import modname` are allowed and required for Astropy, as opposed to absolute (as PEP8 suggests) or the simpler `import modname` syntax. This is primarily due to improved relative import support since PEP8 was developed, and to simplify the process of moving modules.

- General utilities necessary for but not specific to the package or sub-package should be placed in the `packagename.utils` module. These utilities will be moved to the **`astropy.utils`** module when the package is integrated into the core package. If a utility is already present in **`astropy.utils`**, the package should always use that utility instead of re-implementing it in `packagename.utils` module.

LSST

https://confluence.lsstcorp.org/display/LDMDG/Python+Coding+Standard

- when needed use the `__all__` variable as import scheme
- within a project, only use relative imports when importing from another submodule

Modules designed for use via "`from M import *`" SHOULD use the `__all__` mechanism

Modules that are designed for use via "`from M import *`" should use the `__all__` mechanism to ensure only the globals comprising the public API are exported. Failure to use the `__all__` mechanism results in all names in the module's namespace, which do not begin with a single '_', being exported as global.

Relative imports SHOULD be used when importing another module from the same package

Consider this layout:

```
mypkg/
    __init__.py
    foo.py
    bar.py
```

If `foo` wants to import `bar`, the safe way to do this (Python 2.6 and later) is to use relative import:

```
from . import bar
```

Or, if you just want a few symbols from `bar`, this also works:

```
from .bar import thing1, thing2
```

This avoids any danger of name collision with a module on the python path named bar. Relative import statements are richer than suggested by the example; see PEP 328 for details.

THE RULES FOR EUCLID

http://euclid.roe.ac.uk/projects/codeen-users/wiki/User_Cod_Std-pythonstandard

- **organize your code** into thematic modules & submodules
- for each file, **add an** `__all__` **variable** with the list of externally needed methods/classes (some might just be useful internally) - **after imports and before definitions**
- in each `__init__.py` file
  - **add a docstring** to explain the role of the (sub-)module
  - **use relative imports** to import the desired module methods/classes/variables
  - use the `import *` syntax when needed (nowhere else !)
- **use tests** AND **check** your code **interactively**