Python advanced class

Module 2, Modules, packages, and organizing the program code

- Importing modules
- Writing modules
- sys.path amd PYTHONPATH
- Packages
- Docstrings

Importing modules

Importing with `import module`

- When the interpreter reaches an `import module` it looks for a file called `module.py`
- The file is parsed and syntax checked like the script and the executed
- The global symbols are evaluated in a separate namespace
- A module may be imported several times, but it is only read and parsed once.

In [1]:
```
import collections
```

The namespace

- A module is evaluated in a namespace called the same as the module
- After import of the module the classes and functions of the module are available from the main namespace by the module name

In [2]:
```
import collections

print(collections.__name__)
d = collections.deque([1, 2, 3])
print(d)
```

```
collections
deque([1, 2, 3])
```

Importing wih `from module import symbol`

- If only a single symbol is needed from a module the alternative `from module import symbol` may be better
- The symbol is imported directly into the main namespace
- The module namespace is still created but not directly accessable from the main namespace
- It is seen tho import `*` to import every symbol into the main namespace, but it is in general considered a bad idea

In [3]:
```
from copy import deepcopy

L1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L2 = deepcopy(L1)
print(L2)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Importing into a shorter named namespace

- If a module namespace is written many times in a script it can be imported as a shorter name
- This still maintains the separation of namespaces while also shortening the typing

In [4]:
```
import typing as t

data: t.List[ t.Tuple[ float, float, str ] ] | None = None
data = [ (1.2 , 1.7, "Strong"), (4.7, 2.1, "Short") ]
```

Writing modules

Defining classes and functions

- A module is similar to a script file, but most contains definition of classes and functions
- Classes, functions, and variables only relevant for internal pupose should be prepended with a `_`

In [5]:
```
def func1(a, b):
    pass

def func2(x, y):
    pass

def _helper_function():
    pass

_counter = 0
```

The `__name__` variable

- When a file is imported as a module, the variable `__name__` is set to the name of the module
- When a file is runned as a script, the variable `__name__` is set to `__main__`
- This is often used to discriminate between whether a file is runned or imported

In [6]:
```
if __name__ == '__main__':
    print("I am runned as a script")
```

```
I am runned as a script
```

Caching of module parsing

- When a module is imported it is parsed into an internal structure called a parse tree
- To save time this structure is dumped into a cache file in a directory called `__pycache__`
- The name of the file is `module.cpython-3xx.pyc` where `3xx` represent the version of Python, like `311` for version 3.11

```
bash$ ls module.* __pycache__/*
module.py
__pycache__/module.cpython-312.pyc
```

sys.path amd PYTHONPATH

Where do Python look for modules?

- When Python tries to import a module the interpreter search in order in these places:
    - In the directory of the script (or working directory if in interactive session)
    - All directories named in environment variable `PYTHONPATH`
    - In a number of installation dependent or virtual invironment dependent directories
- It is possible to inspect the search path through `sys.path`

In [7]:
```
import sys
for p in sys.path:
    print(p)
```

```
/home/tabac/.pyenv/versions/3.12.2/lib/python312.zip
/home/tabac/.pyenv/versions/3.12.2/lib/python3.12
/home/tabac/.pyenv/versions/3.12.2/lib/python3.12/lib-dynload

/home/tabac/work/python_v/env/lib/python3.12/site-packages
```

Setting the PYTHONPATH variable

- To have the Python interpreter look in specific directories, set the environment variable `PYTHONPATH`
- On POSIX platforms (Linux, MacOS, FreeBSD) the directories are separated by colon
- On Microsoft Windows platforms the directories are separated by semicolon

```
bash$ export
PYTHONPATH='/home/python/modules:/usr/local/lib/python/3.12'
bash$ python -c 'from sys import path; print("\n".join(path))'

/home/python/modules
/usr/local/lib/python/3.12
/home/tabac/.pyenv/versions/3.12.2/lib/python312.zip
/home/tabac/.pyenv/versions/3.12.2/lib/python3.12
/home/tabac/.pyenv/versions/3.12.2/lib/python3.12/lib-dynload
/home/tabac/.local/lib/python3.12/site-packages
/home/tabac/.pyenv/versions/3.12.2/lib/python3.12/site-packages
```

Virtual invironments

- For each project it is often desirable to have a particular set of third-party packages installed
- This is often administrated through so-called a virtual environment for each project
- We will look closer into that i the chapter on project organization

Packages

What is a package?

- Often a rather complex set of classes and functions doesn't fit in a single module
- A set of modules may then be placed in a directory instead
- For this to work the directory must contain a file called `__init__.py`

```
bash$ ls mypackage/*
mypackage/__init__.py  mypackage/mymodule.py
bash$ python -c 'import mypackage.mymodule;
mypackage.mymodule.myfunction()'
This is my function!
```

The `__init__.py` file

- The directory is not considered a package unless it contains a file called `__init__.py`
- The file may be empty, have its own classes and functions, or it may import key symbols from other modules

Relative module import

- Modules can import symbols from other modules in the package
- If the other module is placed in the same directory as the importing module it may be prepended with a dot, `.module`

```
from .mymodule import myfunction

myfunction()
```

The `__all__` list

- In general it is a bad idea to import everything with `*` into a namespace
- But it may be desirable to have a set of symbols to be imported into the top `__init__.py` file
- By setting the variable `__all__` to a list of symbol names it is specified what exactly is imported

```
# in mymodule.py:
__all__ = ['myfunction', 'otherfunction']

# in __init__.py
from .mymodule import *

# in the script
import mypackage as mp
mp.myfunction()
```

Docstrings

What to document

- Docstrings belong to every part of a script or module:
    - To the toplevel of the file
    - To the class
    - To methods and functions
- The docstring should describe the purpose of the part, parameters, and return values

In [8]:
```
#!/usr/bin/env python

"""The module/script docstring"""

class BankAccount:
    """The class docstring"""
    def withdraw(self, amount):
        """The method docstring"""

def make_annual_report(accounts):
    """The function docstring"""
    pass
```

Format suggestions

- The docstring format doesn't have a fixed template
- But there are though som suggestions to follow
    - First line should be a summary of the purpose
    - There should be a blank line afterwards
    - Then parameters and return value

In [9]:
```
def pythagoras(a: float, b: float) -> float:
    """
    Calculates the pythagorian relation of a square triangle

    Parameters:
    a (float): The first side of the triangle
    b (float): the other side of the triangle

    Returns:
    float: The hypotheneuse of the triangle
    """
    return (a ** 2 + b ** 2) ** 0.5
```

The doctest module

- The `doctest` module can run all examples in the docstrings and validate them

In [10]:
```
def pythagoras(a: float, b: float) -> float:
    """
    Calculates the pythagorian relation of a square triangle

    >>> pythagoras(a=3.0, b=4.0)
    5.0
    """
    return (a ** 2 + b ** 2) ** 0.5
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```