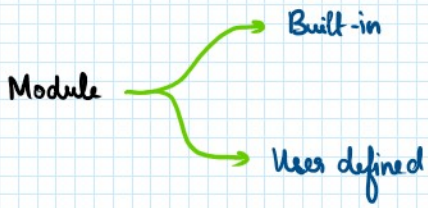
2. Modules

15 November 2023 09:12

MODULE

Any Python file with callable / non-callable statements

To use in another file: import



Characteristics of import

- Runs all executable code in the module when imported
- Imports all of the code which may not be desirable
- Adds module name to symbol table

```
def decx(f):
    def func():
        print("x"*20)
        f()
        print("x"*20)
    return func

def decy(f):
    def func():
        print("y"*20)
        f()
        print("y"*20)
    return func

@decx
@decy     #decx(decy(f1))
def f1():
    print("in f1")

f1()
```

```
import Decorators as dec

dec.f1()
```

```
xxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyy
in f1
yyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyy
in f1
yyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxx
>>>
```

NOTE:
Naming of modules must follow the same rules that are set for identifiers.
Otherwise, error on importing.

NOTE: __pycache__
It is a directory created by the interpreter when a module is imported. It contains the compiled version of the module as a .pyc file, which is used to speed up subsequent imports of the same module. Only for user defined modules
Pycache is updated when the module is changed.

```
from Decorators import f1

f1()
```

→ used to import specific functions only from a module

- Objects that are imported are added to symbol table, not the module
- This implies that you cannot have multiple objects with the same name using "from ... import"

- Objects that are imported are added to symbol table, not the module
- This implies that you cannot have multiple objects with the same name using "from ... import"
  - use aliasing while importing

NOTE: Reload function

```
import importlib
importlib.reload()
```

Used after a module is imported in interpreter mode but some changes have been made to the module post importing.
It updates the __pycache__ file with the latest version of the module.
Only used for interpreter mode.

NOTE: Underscore variables

In Python, variables with a single leading underscore (e.g., _variable) are considered as a convention to indicate that the variable is intended for internal use. It's a signal to other programmers that the variable is part of the internal implementation of a class, module, or other code, and it's not meant to be used directly from outside that scope.

Say you have some _f1() in module_1.

import module_1 module_1._f1() Works	from module_1 import _f1 _f1() Works (as long as there is no clash in identifier names)	from module_1 import * _f1() Error

Using variables with a leading underscore from another module is a matter of convention, and it assumes a level of trust between the developers. It's a signal that the variable is intended for internal use but doesn't strictly enforce any access restrictions.

NOTE : __name__

Returns the name of the file / module that is being executed.

Main program that user is running is referred to as __main__.

Decorators.py - E:/PES/Sem 1/Python/Recursion/Decorators.py (3.9.5)

File Edit Format Run Options Window Help

```
def deco(f):
    def divcheck():
        if n>0:
            return f
        else:
            return "Division not possible."
    return divcheck

@deco  #deco(f1)
def f1(m,n):
    return (m/n)

x="variable"

if __name__=='__main__':
    print(x)
```

```
import Decorators as deco
print(deco.x)
```

```
==========
variable
>>>
```

DOCSTRING

Refer notes on "Strings"

Docstrings can also be accessed by help(module_name.func_name), assuming the function is imported from some module "module_name"

Docstrings can also be added for modules as a whole.

SOME IMPORTANT DISTINGUISHMENTS

Library

- Need not be in Python
- Cannot be edited

Package

- Logical hierarchy of folders
- Can contain libraries

Framework

- More comprehensive set of tools and libraries
- Used as foundation for building specific applications