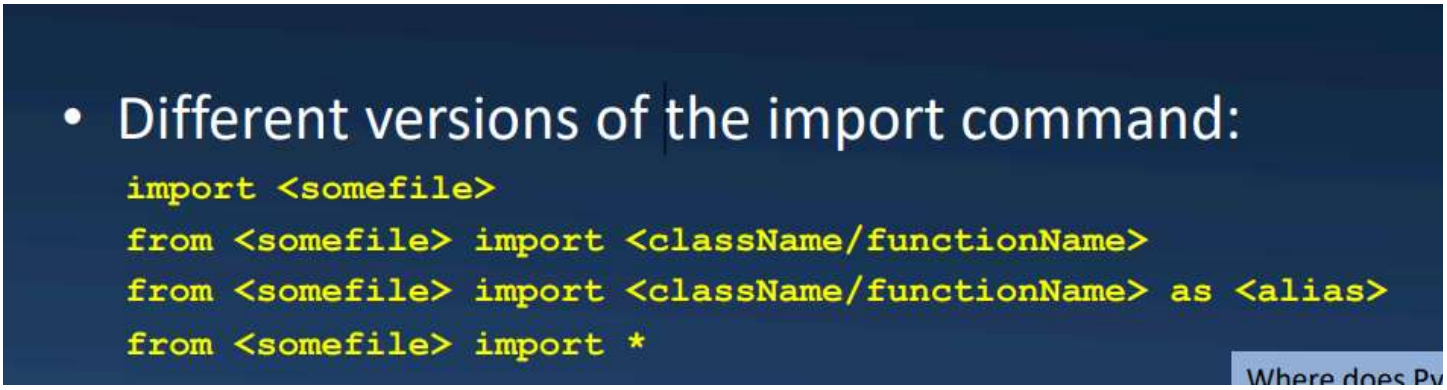
包和模块 Modules and packages



对于小程序，我们可以将所有类放入一个文件中。
随着项目的发展，在众多类中找到一个类可能变得很困难.

而包和模块可以让我们组织起程序
包和模块只是python程序！！！

Module和package的区别：

Module->*.py

Package->Directory{*.py}

在Python中，Modules（模块）和Packages（包）是组织和管理代码的两种方式。

1. **Module（模块）**：
   - 模块是一个包含Python代码的文件，通常以`.py`为扩展名。这些文件包含了函数、类和变量的定义，可以被其他Python程序引入和重用。
   - 每个模块在Python中都有自己的命名空间，这意味着在一个模块中定义的函数、类和变量不会自动影响其他模块。
   - 使用`import`关键字可以在其他Python文件中引入模块，并使用模块中定义的函数、类和变量。
2. **Package（包）**：
   - 包是一种用于组织多个相关模块的方式。它实际上是一个包含了多个模块的文件夹（目录），并且还包含一个特殊的文件`__init__.py`，用于指示Python该文件夹是一个包。
   - 包可以包含子包，即其他文件夹，这些文件夹中也包含了`__init__.py`文件和其他模块文件。
   - 包可以有层级结构，使得代码的组织更加清晰和灵活。

总的来说，模块是一个独立的Python文件，而包是一个包含多个模块的文件夹，并且包还可以包含其他的子包。使用模块和包可以帮助组织和管理大型Python项目的代码，并且使得代码的复用和维护变得更加容易。

Importing Modules and packages

1. `import PIL`
2. `from PIL import Image, ImageMode`
3. `from PIL import Image as im1`

   `from XX import Image as im2`
4. `from PIL import *`

模块引用：
基本引用：

```
import module_name
```

这将导入一个名为module_name的模块，并将其整个内容加载到当前程序中。之后可以使用module_name中定义的函数、类和变量。
使用别名：

```
import module_name as alias
```

这将模块命名为alias，以后可以使用alias代替module_name来引用模块中的内容。
导入特定对象：

```
from module_name import object_name
```

这将只导入模块中的特定对象（函数、类或变量），而不是整个模块。这种方式可以减少命名冲突并提高代码的可读性。
导入所有对象：

```
from module_name import *
```

这将导入模块中的所有对象。尽管方便，但这种方式可能会导致命名冲突，不推荐在生产代码中使用。

包引用：
基本引用：

```
import package_name.module_name
```

这将导入包package_name中的module_name模块。通过点号.来指示模块的层级关系。
使用别名：

```
import package_name.module_name as alias
```

这将模块命名为alias，以后可以使用alias代替package_name.module_name来引用模块中的内容。
从包中导入模块：

```
from package_name import module_name
```

这将从包package_name中导入module_name模块。
从包中导入子包：

```
import package_name.subpackage_name.module_name
```

这将从包package_name的子包subpackage_name中导入module_name模块。
相对导入：
在一个包中的模块中，可以使用相对导入来引用同一包中的其他模块：

```
from . import module_name
```

假设我们有以下项目结构：

go Copy code

```
my_project/
├── package/
|   ├── __init__.py
|   ├── module1.py
|   └── module2.py
└── script.py
```

在这个示例项目中，`my_project`是项目的根目录，`package`是一个包，其中包含了两个模块`module1.py`和`module2.py`，以及一个空的`__init__.py`文件，标识该目录为一个包。`script.py`是一个独立的脚本文件。

绝对导入 vs 相对导入

首先，我们来看一下绝对导入和相对导入的区别。

- **绝对导入**：基于整个Python路径搜索来导入模块。例如 `import package.module1`
- **相对导入**：基于当前模块的位置来导入模块。例如在 `module1.py` 中，我们可以使用相对导入 `from . import module2`

相对导入的语法

相对导入使用点号来表示当前包和模块的位置。点号 `.` 表示当前目录或当前模块，而点点 `..` 表示上级目录或上级模块。

在模块内部，使用以下语法进行相对导入：

```python
from . import module_name
```

或者

```python
from .module_name import some_function
```

在包内的模块中，`from .` 表示相对于当前包的位置进行导入。

在包外的模块中，相对导入使用的是相对于调用脚本的位置。所以，包外的脚本 `script.py` 可以这样导入 `package` 中的模块：

```python
from package import module1
```

让我们来看看如何在示例项目中使用相对导入。

假设我们想在 `module1.py` 中导入 `module2.py` 中的函数。

python Copy code

```python
# module1.py

from . import module2

def function_in_module1():
    module2.function_in_module2()
```

在这里，`from . import module2` 表示从当前包中导入 `module2` 模块。

在 `script.py` 中，我们可以这样导入 `module1` 中的函数：

python Copy code

```python
# script.py

from package import module1

module1.function_in_module1()
```

这样，就可以使用相对导入在Pyth[illegible]项目中组织代码，并避免命名冲突。

这样导入的话，如果当前我文件已经有了Database，那么当前文件的Database会被覆盖

student.py

```
...
from database import *
db = Database()
# Do queries on db
```

Import all classes and functions from the `database` module into the `student` namespace.

Don't do this. Every experienced Python programmer will tell you that you should never use this syntax. Makes code impossible to manage and maintain.