

[笔记][LIKE-Python-6][02]

Python

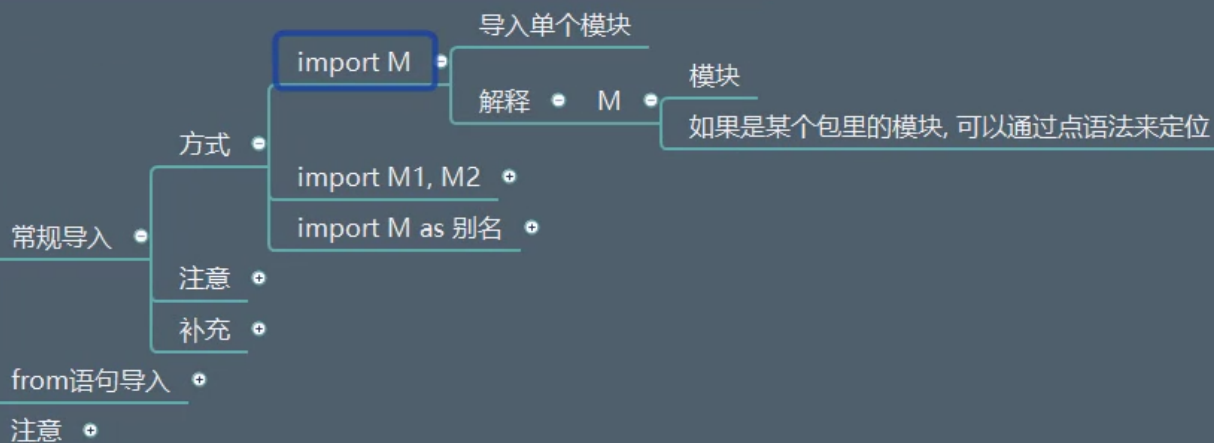
[笔记][LIKE-Python-6][02]

- 007. 包和模块的导入-常规导入-导入单个包
- 008. 包和模块的导入-常规导入-导入多个包
- 009. 包和模块的导入-常规导入-导入包的同时起个别名
- 010. 包和模块的导入-常规导入-注意和补充
- 011. 包和模块的导入-from导入-sz自创理解法
- 012. 包和模块的导入-from导入-从包导模块
- 013. 包和模块的导入-from导入-从模块导资源
- 014. 包和模块的导入-from导入-注意事项
- 015. 包和模块的导入-from导入-导入特例
- 016. 包和模块的导入-导入模块底层做的事
- 017. 包和模块的导入-导入模块底层做的事-结论
- 018. 包和模块的导入-模块检索路径-内置和sys.path
- 019. 包和模块的导入-模块检索路径-sys.path组成
- 020. 包和模块的导入-模块检索路径-修改方式1
- 021. 包和模块的导入-模块检索路径-修改方式2
- 022. 包和模块的导入-模块检索路径-修改方式3
- 023. 包和模块的导入-模块检索路径-第二次导入
- 024. 包和模块的导入-导入场景-局部导入
- 025. 包和模块的导入-导入场景-覆盖导入
- 026. 包和模块的导入-导入场景-循环导入
- 027. 包和模块的导入-导入场景-可选导入
- 028. 包和模块的导入-导入场景-包内导入-上
- 029. 包和模块的导入-导入场景-包内导入-中
- 030. 包和模块的导入-导入场景-包内导入-下

007. 包和模块的导入-常规导入-导入单个包

友情提醒 • 这一块知识点本身比较碎; 需要自己多做实验慢慢体会掌握

理论基础 • 导入模块的作用 • 可以使用这个模块里面的内容



008. 包和模块的导入-常规导入-导入多个包

导入多个包

- 写多条 `import` 语句
- 用逗号分隔 `import a, b`

009. 包和模块的导入-常规导入-导入包的同时起个别名

导入包的同时起别名

```
import module as m
```

应用场景

简化资源访问前缀

```
if file_extension == "txt":
    import txt_parse
    txt_parse.open()
    txt_parse.read()
    txt_parse.close()
elif file_extension == "doc":
    import doc_parse
    doc_parse.open()
    doc_parse.read()
    doc_parse.close()
```

增加程序的扩展性 例如

```
if file_extension == "txt":
    import txt_parse as parse
elif file_extension == "doc":
    import doc_parse as parse
```

优化后

```
parse.open()
parse.read()
parse.close()
```

010. 包和模块的导入-常规导入-注意和补充

实际上 Python 把包和模块也当成了对象，使用 `.` 的语法来访问

注意 使用时，需要指明资源的模块名称 例如： `p1.my_moudle.run()`

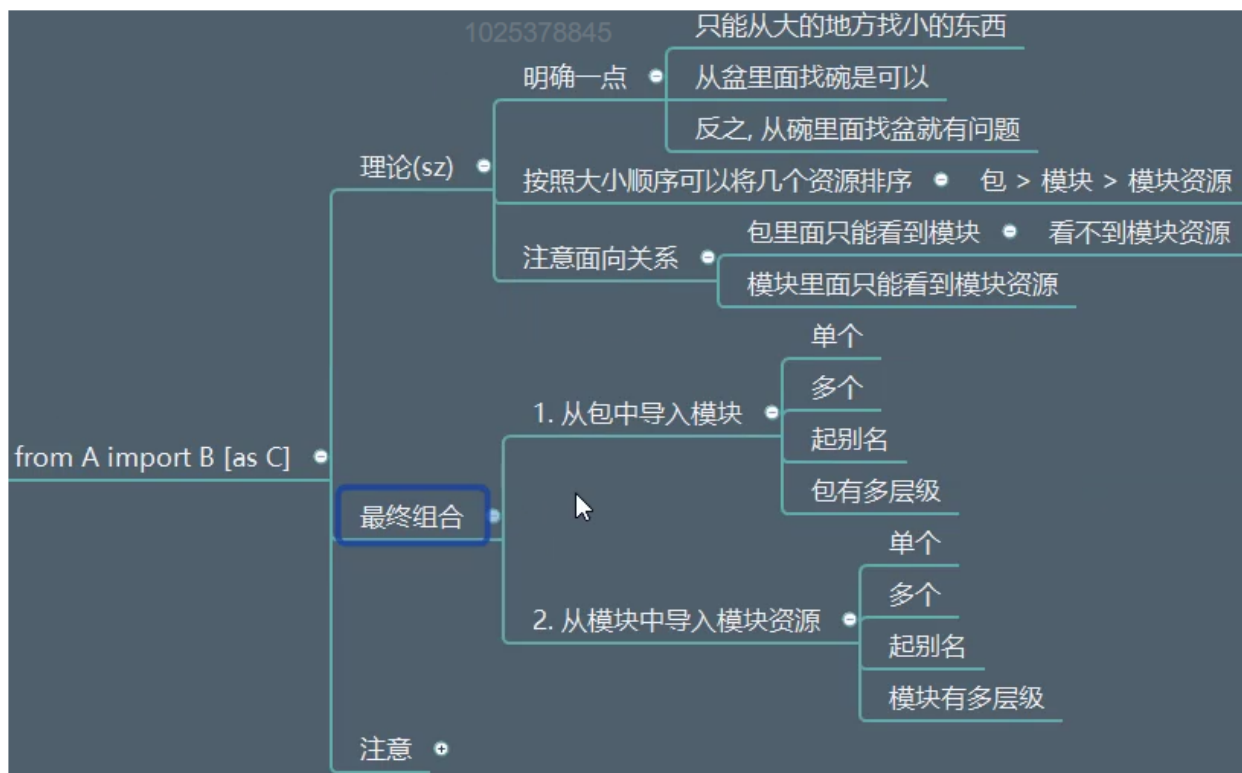
默认不会导入任何模块

补充 如果导入的是一个包

解决方案

1. 在 `__init__` 文件中，再次导入需要的模块
2. 应该以 `from ... import ...` 的形式导入

011. 包和模块的导入-from导入-sz自创理解法



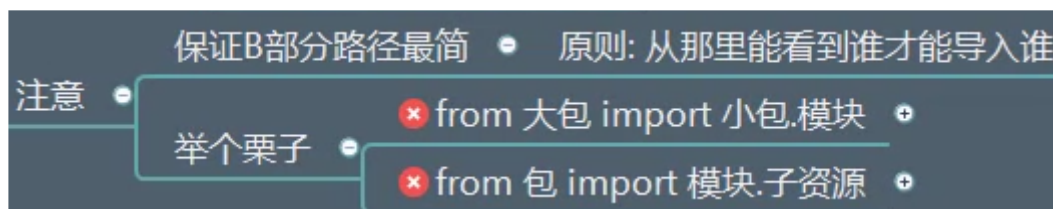
012. 包和模块的导入-from导入-从包导模块

from 导入注意

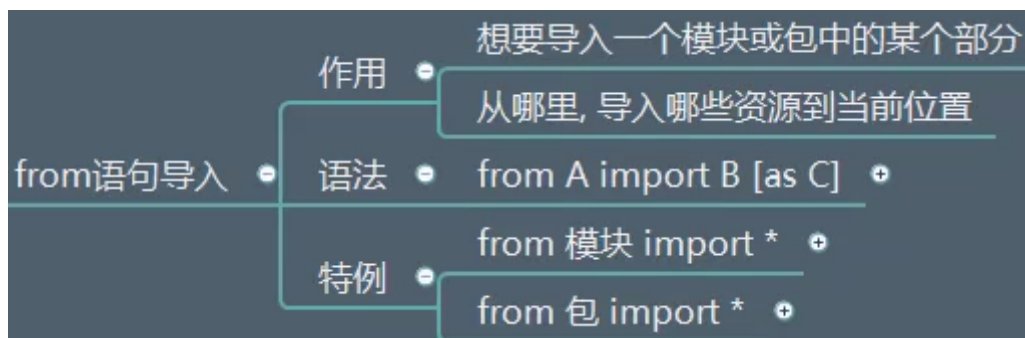
- 不管是 `from` 导入还是直接 `import` 导入, 导入之后使用的时候, 只用 `import` 后面导入的东西来访问资源
- `from` 导入要保证 `import` 后面的东西最简化

013. 包和模块的导入-from导入-从模块导资源

014. 包和模块的导入-from导入-注意事项



015. 包和模块的导入-from导入-导入特例



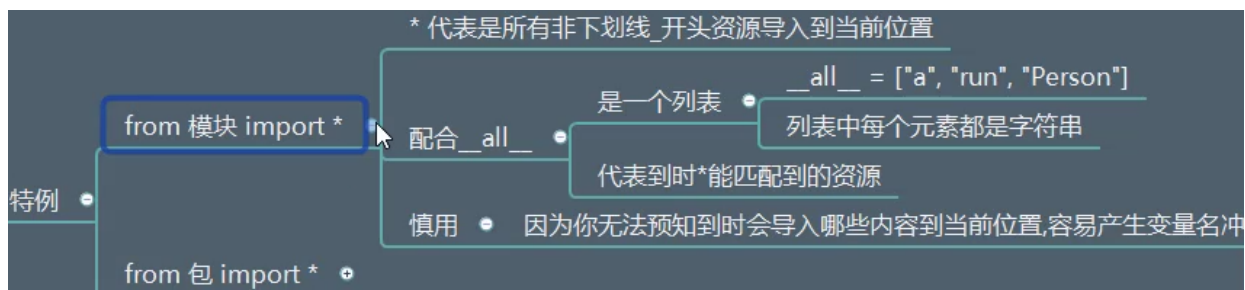
模块的 `__all__` 变量

- 在模块里面放 `__all__` 变量, 它是一个由字符串组成的列表。
`__all__ = ['num1', 'num2']`
- 使用 `from ... import *` 的时候可以导入里面的所有内容。
- `__all__` 变量的位置可以随便放。

包的 `__all__` 变量

- 比如 `__all__ = ['模块1', '模块2']`

如果没有 `__all__` 变量, 会导入所有资源, 除了以下划线开头的资源。
包括单下划线和双下划线开头的资源。



016. 包和模块的导入-导入模块底层做的事

注意	1. 导入模块后具体做了什么事？
	2. 从哪个位置找到需要导入的模块？
	3. 导入模块的常见场景？

第一次导入时	1. 在自己当下的命名空间中, 执行所有代码
	2. 创建一个模块对象, 并将模块内所有顶级变量以属性的形式绑定在模块对象上
	3. 在import的位置, 引入import后面的变量名称到当前命名空间
第二次导入时	
结论	

```
import tools
```

```
# 查看模块对象
```

```
print(tools)
```

```
# <module 'tools' from 'D:\\code\\likepython\\6\\tools.py'>
```

```
# 查看内存地址
```

```
print('内存地址为', id(tools))
```

```
# 内存地址为 1511359636648
```

```
# 查看类型
```

```
print('类型为', type(tools))
```

```
# 类型为 <class 'module'>
```

```
# 查看模块对象的方法和属性
```

```
print('模块对象的方法和属性为', tools.__dict__)
```

jpch89：在自己当下的命名空间中，执行所有代码

这个**自己当下的命名空间**指的是被导入模块的命名空间，还是导入模块的命名空间？

老师后来讲了：**是在被导入模块所在的命名空间执行代码**

假如说使用了被导入模块的函数，该函数也是在被导入模块中的命名空间执行的。

注意：

第二次导入的时候，并不会重新新建一个模块对象

`print(id(module))` 得到的 `id` 是一样的

017. 包和模块的导入-导入模块底层做的事-结论

导入模块后究竟做了什么事情？

第一次导入时	1. 在自己当下的命名空间中, 执行所有代码
	2. 创建一个模块对象, 并将模块内所有顶级变量以属性的形式绑定在模块对象上
	3. 在import的位置, 引入import后面的变量名称到当前命名空间
第二次导入时	直接执行上述第3步
注意: 两种导入方式都会大致执行以上的步骤	
结论	1. 多次导入模块, 该模块并不会执行多次
	2. 两种导入方式不存在哪一种更省内存 • 区别在于把哪一部分内容拿到当前位置来用

注意

- 使用 `from module import something` 这种方式, 导入部分资源, 也是会执行被导入模块的所有代码。
- 多次导入不会多次执行。

面试题

两种导入方式并不会存在哪种更节省内存。

所有对象都被创建了, 区别在于拿哪些内容到当前位置来用而已。

018. 包和模块的导入-模块检索路径-内置和sys.path

从哪个位置找到需要导入的模块?

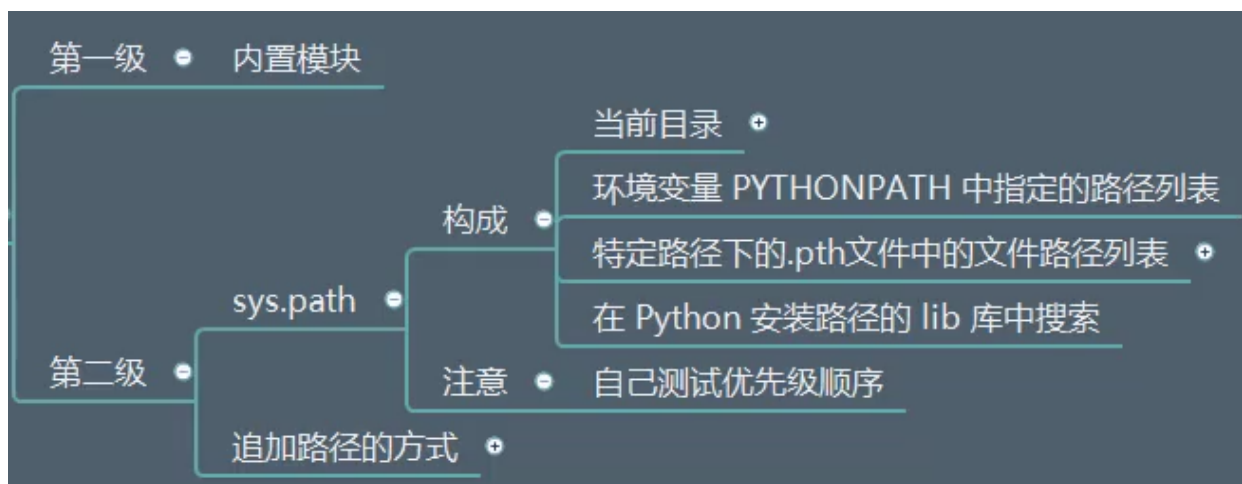
- 第一次导入: 按照模块检索路径顺序去找。
 - 第一级: 内置模块
 - 第二级: `sys.path`
- 第二次导入

 <code>math</code>	<code><built-in></code>
 <code>cmath</code>	<code><built-in></code>
 <code>fnmatch</code>	<code>C:\Python\36\Lib</code>
 <code>formatter</code>	<code>C:\Python\36\Lib</code>

后面有 `<built-in>` 都是内置模块, 优先级最高



019. 包和模块的导入-模块检索路径-sys.path组成



`sys.path` 构成：

- 当前目录
- 环境变量 `PYTHONPATH` 中指定的路径列表
- 特定路径下 `.pth` 文件中的路径列表
- 在 `Python` 安装路径以及该路径下的 `lib` 库中搜索

`sys.path` 优先级顺序

- 当前目录
- 环境变量 `PYTHONPATH` 中指定的路径列表
- `Python` 安装路径
- `Python` 安装路径下的 `.pth` 文件中的路径
- `Python` 安装路径的 `lib` 库 (`lib\site-packages`)
- `lib` 库中的 `.pth` 文件中的路径

注意

- 自己测试优先级顺序

020. 包和模块的导入-模块检索路径-修改方式1

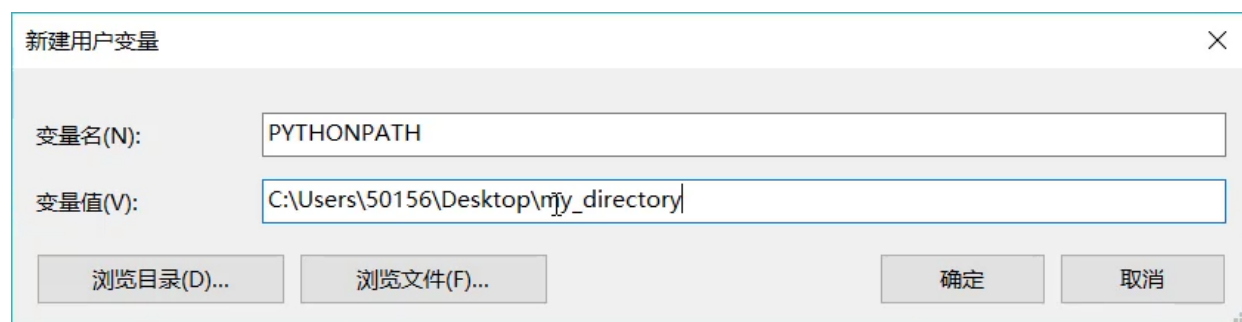
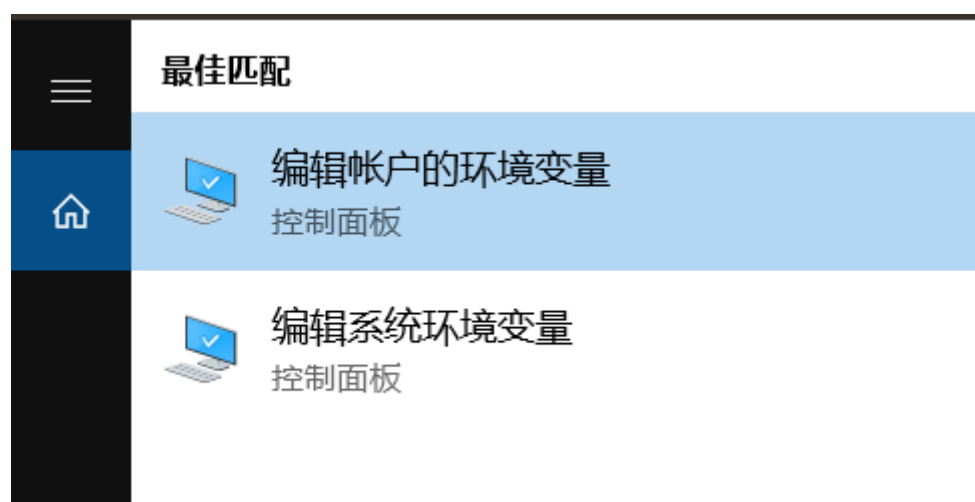


直接修改 `sys.path`

```
import sys
print(sys.path)
sys.path.append(r'新增路径')
```

021. 包和模块的导入-模块检索路径-修改方式2

win10 可以在开始菜单中直接搜索 环境

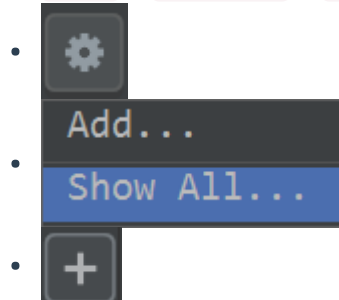



注意

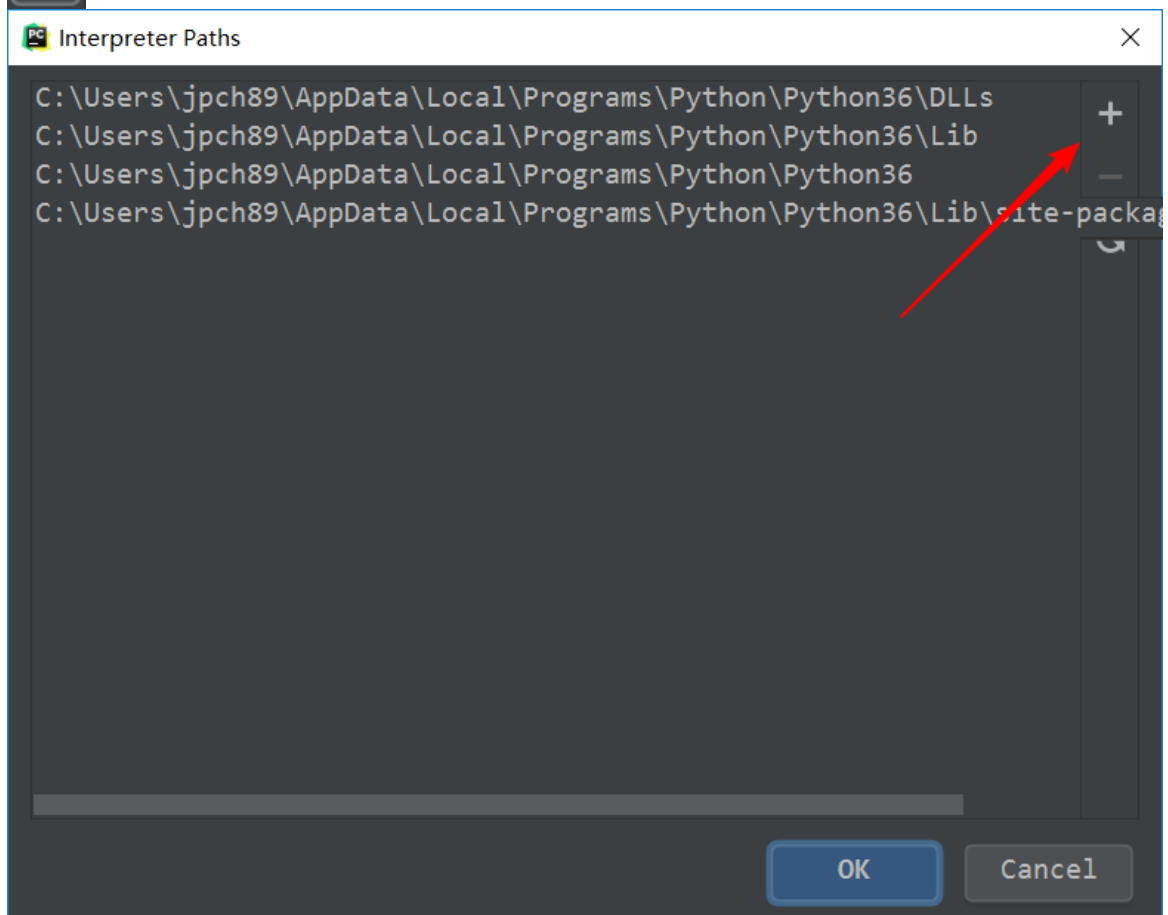
- 使用这种方式仅仅在 `cmd` 下面有效，`PyCharm` 中会有问题

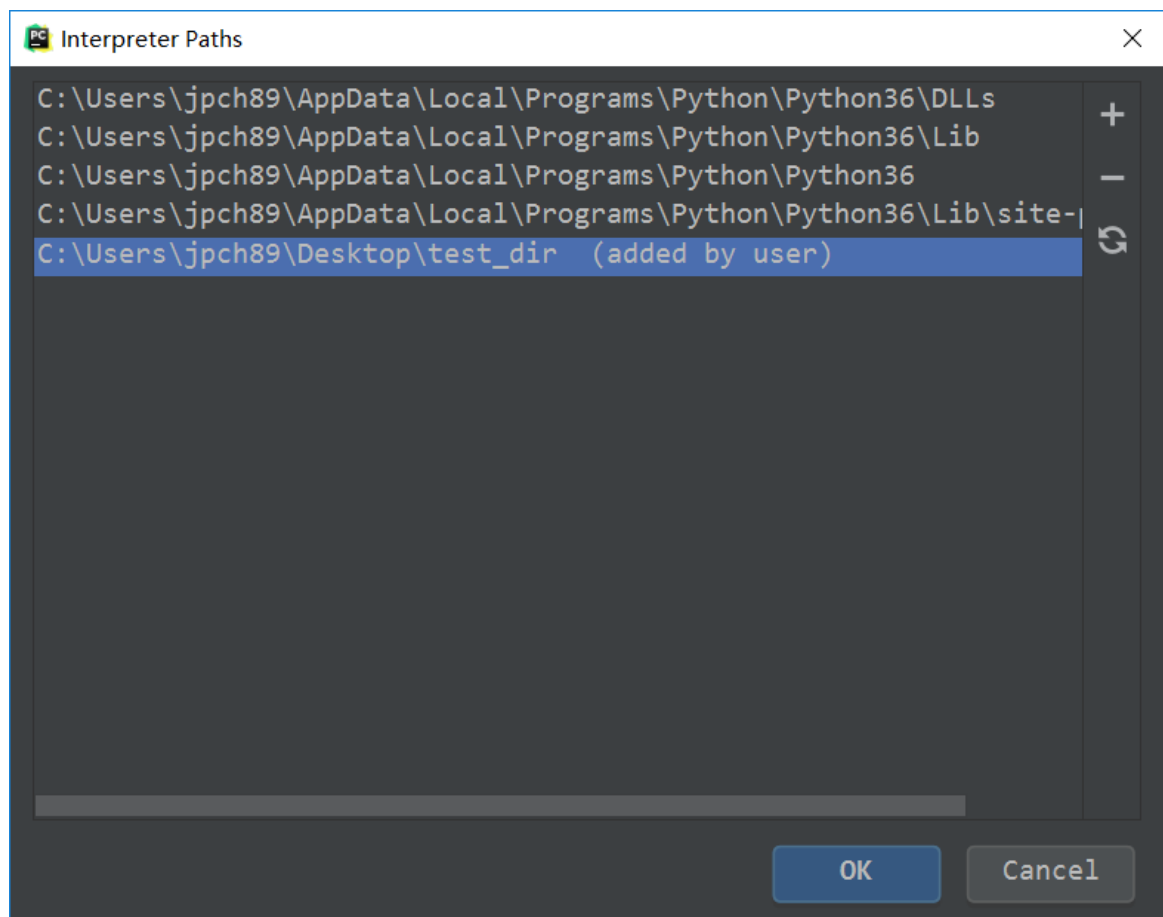
PyCharm 修正方式

- `File` - `Settings` - `Project` - `Project Interpreter`



- Show paths for the selected interpreter 





这样就可以使用了。

022. 包和模块的导入-模块检索路径-修改方式3

在哪里创建 `.pth` 文件

```
import site
print(site.getsitepackages())
```

得到结果

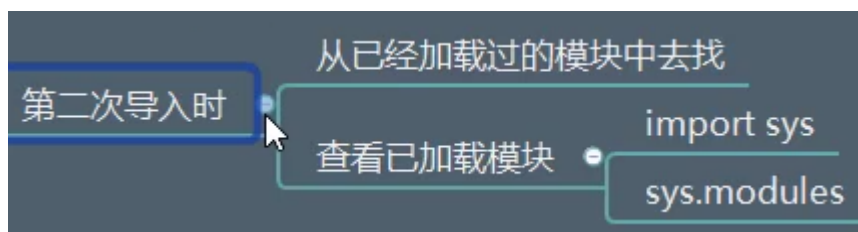
```
['C:\\Users\\jpch89\\AppData\\Local\\Programs\\Python\\Python36', 'C:\\Users\\jpch89\\AppData\\Local\\Programs\\Python\\Python36\\lib\\site-packages']
```

- 在 `Python` 的安装路径
- 在 `Python` 安装路径中的 `lib/site-packages` 中

`.pth` 文件怎么写

- 路径直接写
- 回车换行隔开

023. 包和模块的导入-模块检索路径-第二次导入

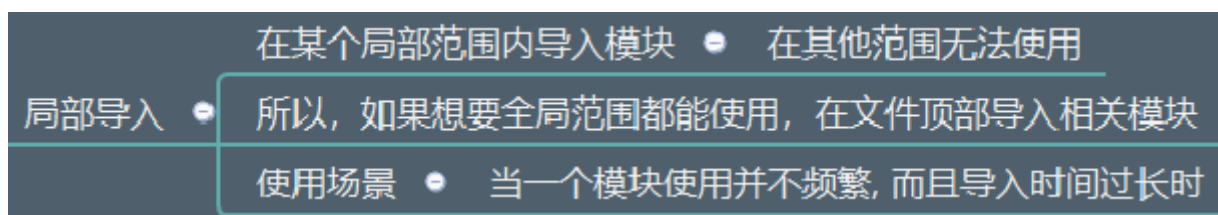


```
import sys
print(sys.modules)
```

024. 包和模块的导入-导入场景-局部导入

导入场景

- 局部导入
- 覆盖导入
- 循环导入
- 可选导入
- 包内导入



有关命名空间

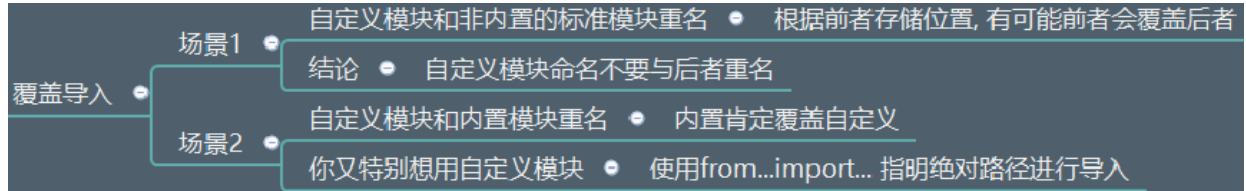
- `if`、`for`、`while` 函数都不会产生一个新的命名空间。
- 函数、类、模块可以产生！

局部导入示例

```
def run():
    import other
    print(other.name)
```

```
print(other.age)
```

025. 包和模块的导入-导入场景-覆盖导入



绝对路径要写全, 比如

```
from 包和模块 import sys
```

026. 包和模块的导入-导入场景-循环导入



test.py

```
import other

print(other.o1)
print(other.o2)

t1 = 't1'
t2 = 't2'
```

other.py

```
import test

print(test.t1)
print(test.t2)

o1 = 'o1'
o2 = 'o2'
```

运行 test.py 流程分析:

- 解释器碰到 `test.py` 中的 `import other` 语句
- 查看 `sys.modules` 里面有没有名为 `other` 的模块
- 发现没有，所以这是**第一次导入**，根据模块查找路径在当前文件夹下找到 `other.py`，把 `other` 模块加入到 `sys.modules` 字典中
- 运行 `other.py`，在 `other.py` 中碰到 `import test` 语句
- 查看 `sys.modules`，发现**没有**名为 `test` 的模块，所以这是**第一次导入**，根据模块查找路径在当前文件夹下找到 `test.py`，把 `test` 模块名加入到 `sys.modules` 字典中
- 运行 `test.py`，碰见 `import other` 语句
- 但是此时 `sys.modules` 中已经有了 `other` 模块，所以不会执行 `other.py`，只是把 `other` 这个名字拿到当前的位置，然后继续执行下一句 `print(other.o1)`
- 此时 `other` 里面**还没有创建** `o1` 这个变量
- 所以会**报错**，提示 `other` 模块没有名为 `o1` 的属性！

```
AttributeError: module 'other' has no attribute 'o1'
```

老师说可以从结构设计层面来避免循环引用。

027. 包和模块的导入-导入场景-可选导入

可选导入	概念	两个功能相近的包, 根据需求优先选择其中一个导入
	场景	有两个包A和B都可以实现相同的功能 想优先使用A; 而且需要做到在没有A的情况下, 使用B做备选
	实现	使用try...except...进行实现

优先导入 `other1` 然后导入 `other2`

```
try:
    import other1 as o
except ModuleNotFoundError:
    import other2 as o

# 在后面统一使用 o 来调用
```

注意：适用于两个模块变量和 `API` 相同的时候

028. 包和模块的导入-导入场景-包内导入-上

包内导入可以分为：

- 绝对导入
- 相对导入

绝对导入和相对导入是针对**包内导入**来讲的。

理论

Python 相对导入与绝对导入，这两个概念是相对于包内导入而言的
包内导入即是包内的模块导入包内部的模块

注意

- 当我们尝试用解释器执行某一个 `py` 文件的时候，那他就会确定当前文件所在的目录，再把目录添加到 `sys.path` 中
- 添加过之后，再往后，基本上 `sys.path` 里面的内容已经确定

029. 包和模块的导入-导入场景-包内导入-中

使用 `print(__name__)` 查看当前所在模块的模块名称

- 如果直接以脚本运行，`__name__` 里面存的是 `__main__`
- 如果是模块形式加载，`__name__` 的名称由加载路径决定，比如 `包名.子包名.模块名`。把最前面那部分，即 `包名` 称为顶级名称。

假如一个 `py` 文件以脚本形式运行，不能用 `from . import xxx`，因为此时该文件的 `__name__` 为 `__main__`，所以 `.` 没有与之对应的顶级名称。

030. 包和模块的导入-导入场景-包内导入-下

在 `Python 2` 中，默认采用的是相对导入，把不带 `.` 和 `..` 的称为隐式的相对导入。

这样不好的地方是如果自定义模块与内置模块重名，会先导入自定义模块，这样就没有办法使用内置模块了。

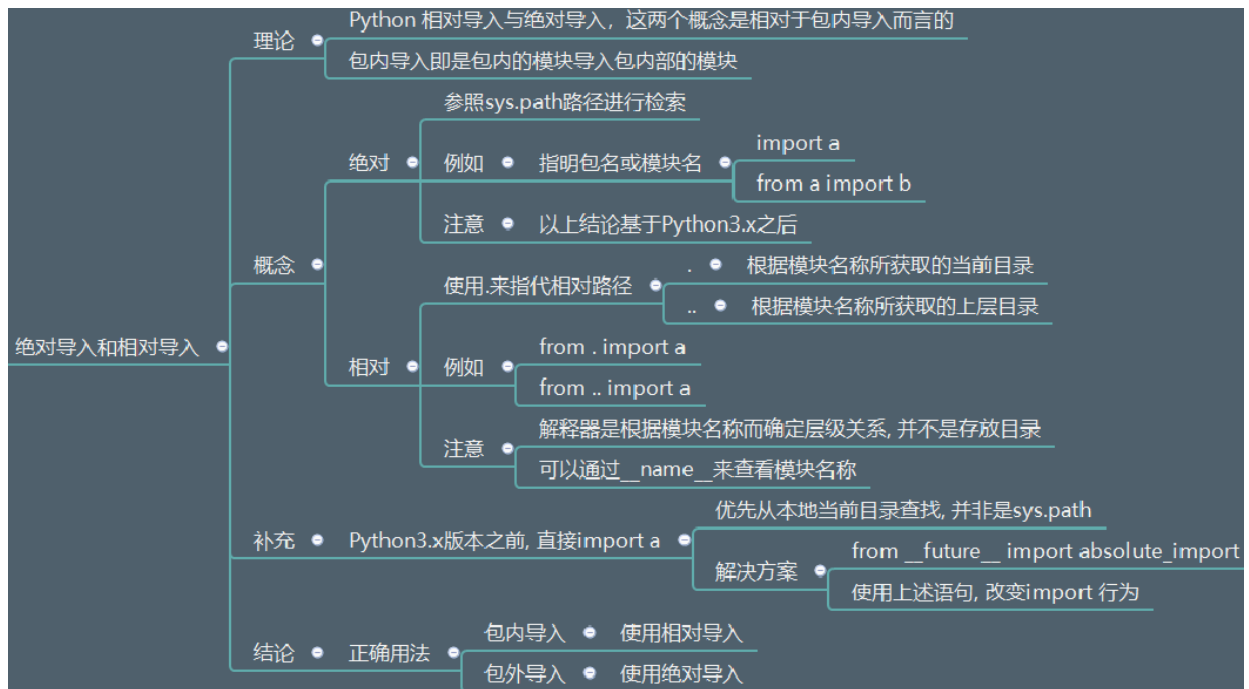
解决方案是在 `Python 2` 中禁用掉隐式相对导入，改用绝对导入。

```
from __future__ import absolute_import
```

结论

- 包内模块之间使用相对路径

- 包外导入包内模块，用绝对路径



Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

注意相对导入是基于当前模块名字的导入。因为主模块的名字总是 `"__main__"`，所以作为程序主模块的文件必须要使用绝对导入。

<https://docs.python.org/3.7/tutorial/modules.html>

补充：始终用 `from ... import ...` 形式的绝对导入也可以避免出错！

完成于 201811102038