

# 模块

黄书剑





# 从语句构建更大规模的代码

- 通过复合语句控制程序流程
  - 条件分支语句 if-else if-elif-else 等
  - 循环语句 while for 等
  - 函数、类定义等
- 更大规模的模块、包、库等
- 在python中:
  - 模块：单个文件
  - 包：单个目录（包含若干模块或者子包）
  - 库：相关的若干包的组合



# Hello World!

- 第一个python程序!
- 交互方式 v.s. 文件方式

在python交互窗口中输入:

```
>>> print ("hello world")
hello world
>>>
```

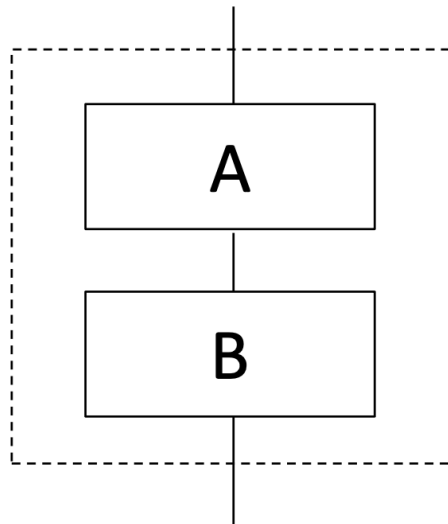
文件python1.py:

```
print ("hello world")
```

```
huangsj$ python3 python1.py
hello world
huangsj$
```

# 单文件中的程序执行

- 语句的默认执行顺序为顺序执行
  - 赋值、函数调用、表达式运算、函数定义、类定义
- 改变运行顺序需要依赖特殊的控制规则（复合语句）
  - 分支、循环（、异常处理）



交互式运行时解释器顺序执行每个输入语句

以文件输入时解释器顺序执行文件中的每个语句

计算模型：依次解释每一个语句（名绑定、求值）



## 单文件中的程序执行

- 每一个单独的文件被看做一个python模块 (module)
  - 每个模块包含自己的变量、函数、可执行代码等
- 内建函数 `dir()` 可以用于查看模块中的绑定关系
- 变量 `__name__` 存储当前模块的名字, 可用于区分不同模块
- 文件当做脚本运行时, 可以通过 `sys.argv` 获取控制台的参数

*test.py*

```
import sys  
print(sys.argv)
```

```
huangshujian$ python test.py abc xyz  
['test.py', 'abc', 'xyz']
```



## 多文件中的程序执行

- 在当前文件中引入其他文件中定义的绑定关系或计算过程
  - 使用import语句，引入其他模块

```
import test_print  
print(test_print.__name__)
```

*test\_print.py*

```
def printID(x):  
    print(x, ":", id(x))  
  
def work_print(x):  
    x += 5  
    printID(x)
```

```
C++:  
#include "myfunc.h"
```



# 导入模块

- 特别的，当import一个模块时，将会顺序引入文件所有内容（与执行相同）
  - 包括函数、变量的绑定
  - 包括部分执行代码（测试代码等）
  - 所有绑定关系被记录在该模块自己的符号表中

```
def printID(x):  
    print(x, ":", id(x))  
  
def work_print(x):  
    x += 5  
    printID(x)  
  
x = 3  
printID(x)  
work_print(x)  
printID(x)
```

如何处理导入模块和当前模块的不同绑定关系？  
如何处理导入模块和当前模块中的执行代码？

C++:  
每个可执行文件有唯一的  
main函数作为程序入口



# 模块的名空间

- 对于引入模块中的名绑定
  - 创建一个引入模块的名空间

```
import test_print  
  
x = 10  
  
test_print.work_print(x)  
print(test_print.x)  
print(x)
```

*test\_print.py*

```
def printID(x):  
    print(x, ":", id(x))  
  
def work_print(x):  
    x += 5  
    printID(x)  
  
x = 8
```





# 模块中的执行代码

- 对于引入模块中的执行代码
  - 使用\_\_name\_\_区分模块引入和执行

```
import test_print  
  
x = 10  
test_print.work_print(x)
```

*test\_print.py*

```
def printID(x):  
    print(x, ":", id(x))  
  
def work_print(x):  
    x += 5  
    printID(x)  
  
x = 8  
print(x)
```



## 模块中的执行代码

- 对于引入模块中的执行代码
  - 使用\_\_name\_\_区分模块引入和执行
  - 当前模块被直接执行时值为“\_\_main\_\_”，否则为模块名

```
import test_print
x = 10
test_print.work_print(x)
print(__name__)
```

- 此时，文件既能被import也能当做脚本使用

*test\_print.py*

```
def printID(x):
    print(x, ":", id(x))

def work_print(x):
    x += 5
    printID(x)

if __name__ == "__main__":
    x = 8
    print(x)
```



## 多种import语句

- 使用import语句，引入其他模块

- 执行被引入模块中的全部代码
- 创建一个引入模块的名空间

```
import <module> [as <name>]
```

- 可以仅将部分模块中的元素导入当前模块

- 被导入的元素将在当前frame中进行绑定

```
from <module> import <element> [as <name>]
```


- as name用于在当前frame创建新的名绑定

## 包：组织多个文件/模块

- 一个包是一个目录，由多个模块或者子包构成
- 引入包
  - 执行目录下的\_\_init\_\_.py文件

```
package_a/__init__.py
```

```
import package_a.module_a
```




```
package_a/  
  __init__.py  
  module_a.py  
  subpackage_b/  
    __init__.py  
    module_b1.py  
    module_b2.py  
  subpackage_c/  
    __init__.py  
    module_c1.py  
    module_c2.py  
  ...
```

在\_\_init\_\_.py中引入可能会被外部使用的子包、模块、函数、变量等

## 包：组织多个文件/模块

- 一个包是一个目录，由多个模块或者子包构成
- 引入包
- 引入包中指定模块

```
import package_a.subpackage_b  
from package_a.subpackage_c import module_c1
```



```
package_a/  
  __init__.py  
  module_a.py  
  subpackage_b/  
    __init__.py  
    module_b1.py  
    module_b2.py  
  subpackage_c/  
    __init__.py  
    module_c1.py  
    module_c2.py  
  ...
```

## 包：组织多个文件/模块

- 一个包是一个目录，由多个模块或者子包构成
- 引入包
- 引入包中指定模块
- 引入包中“全部”内容

```
from <package> import *
```

– 使用\_\_all\_\_约定全部会被外部使用的符号

```
package_a/__init__.py
```

```
__all__ = ['module_a']
```

```
package_a/  
__init__.py  
module_a.py  
subpackage_b/  
__init__.py  
module_b1.py  
module_b2.py  
subpackage_c/  
__init__.py  
module_c1.py  
module_c2.py  
...
```

```
import package_a.subpackage_b.module_b1  
__all__ = ['module_a', 'subpackage_b', 'subpackage_b.module_b1']
```



# 模块与抽象

- 模块内部实现的封装
  - 使用“\_”前缀定义仅在当前包内使用的变量或函数
  - 使用“\_\_init\_\_.py”控制向外导出的变量或函数
- 向外提供某种函数的定义和实现
- 向外提供某种类的定义和实现



## 兼顾脚本的特性和运行效率

- 包也可以作为运行的单元
  - 执行包中的\_\_main\_\_.py
- 同一个模块仅会被加载一次
- 模块会被编译后进行缓存以便快速加载
  - \_\_pycache\_\_目录下的 \*.pyc 文件
- 可以使用-o -oo进行进一步优化，提升模块的执行速度



- Python中组织大规模程序的方法
  - 函数、类、模块、包
- 部分可能有用的信息
  - `sys.argv`
  - python文档中关于模块的介绍  
<https://docs.python.org/3/tutorial/modules.html>