[笔记][黑马 Python 之 Python 面向对象 - 3]

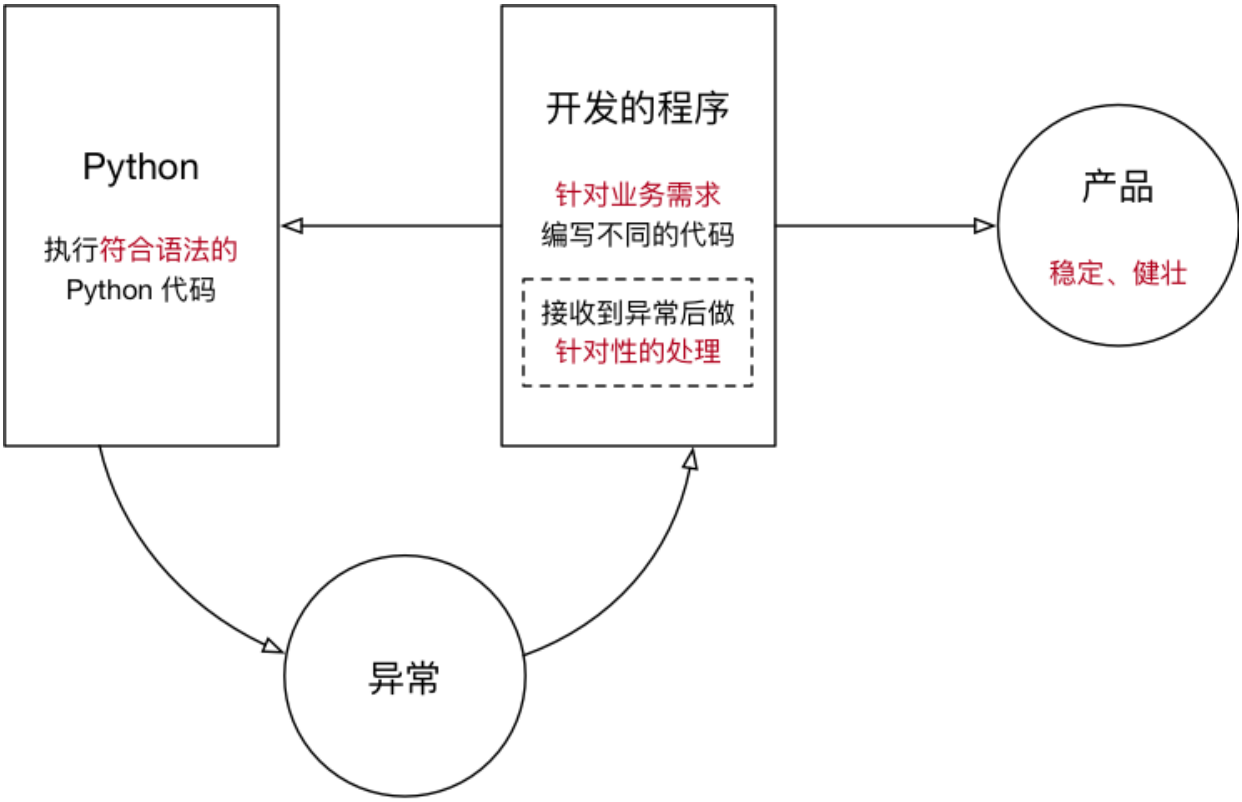
Python

099. eval-01-基本使用
100. eval-02-[扩展]不要直接转换input结果

---

064. 异常-01-异常的概念以及抛出异常的原因

- 程序在运行时，如果 `Python` 解释器 **遇到** 到一个错误，**会停止程序的执行，并且提示一些错误信息**，这就是 **异常**
- **程序停止执行并且提示错误信息** 这个动作，我们通常称之为：**抛出(raise)异常**



程序运行时，如果遇到错误，就会抛出异常

> 程序开发时，很难将 **所有的特殊情况** 都处理的面面俱到，通过 **异常捕获** 可以针对突发事件做集中的处理，从而保证程序的 **稳定性和健壮性**

---

065. 异常-02-简单的异常捕获

简单的捕获异常语法

- 在程序开发中，如果 **对某些代码的执行不能确认是否正确**，可以增加 `try`(尝试) 来 **捕获异常**
- 捕获异常最简单的语法格式：

```
try:
    尝试执行的代码
except:
    出现错误的处理
```

- `try` **尝试**，下方编写**要尝试代码**，**不确定是否能够正常执行的代码**
- `except` 如果不是，下方编写**假如尝试失败的代码**

```
try:
    # 不能确定正确执行的代码
    num = int(input('请输入一个整数：'))
except:
    # 错误的处理代码
    print('请输入正确的整数')

print('-' * 50)
```

---

066. 异常-03-根据错误类型捕获异常

- 在程序执行时，可能会遇到 **不同类型的异常**，并且需要 **针对不同类型的异常**，**做出不同的响应**，这个时候，就需要捕获错误类型了
- 语法如下：

```
try:
    # 尝试执行的代码
    pass
except 错误类型1:
    # 针对错误类型1，对应的代码处理
    pass
except (错误类型2, 错误类型3):
    # 针对错误类型2 和 3，对应的代码处理
    pass
except Exception as result:
    print("未知错误 %s" % result)
```

- 当 `Python` 解释器 **抛出异常** 时，**最后一行错误信息的第一个单词，就是错误类型**

```
try:
```

```
    # 提示用户输入一个整数
    num = int(input('输入一个整数：'))
    # 使用 8 除以用户输入的整数并输出
    result = 8 / num
    print(result)
except ZeroDivisionError:
    print('除0错误')
except ValueError:
    print('请输入正确的整数')
```

---

067. 异常-04-捕获未知错误

捕获未知错误

- 在开发时，要预判到所有可能出现的错误，还是有一定难度的
- 如果希望程序 **无论出现任何错误**，都不会因为 `Python` 解释器 **抛出异常而被终止**，可以再增加一个 `except`

语法如下：

```
except Exception as result:
    print("未知错误 %s" % result)
```

示例：

```
try:
    # 提示用户输入一个整数
    num = int(input('输入一个整数：'))
    # 使用 8 除以用户输入的整数并输出
    result = 8 / num
    print(result)
except ValueError:
    print('请输入正确的整数')
except Exception as e:
    print('未知错误 %s' % e)
```

---

068. 异常-05-异常捕获的完整语法

- 在实际开发中，为了能够处理复杂的异常情况，完整的异常语法如下：

```
try:
    # 尝试执行的代码
    pass
except 错误类型1:
    # 针对错误类型1，对应的代码处理
    pass
except 错误类型2:
    # 针对错误类型2，对应的代码处理
    pass
except (错误类型3, 错误类型4):
    # 针对错误类型3 和 4，对应的代码处理
    pass
except Exception as result:
    # 打印错误信息
    print(result)
else:
    # 没有异常才会执行的代码
    pass
finally:
    # 无论是否有异常，都会执行的代码
    print("无论是否有异常，都会执行的代码")
```

- `else` 只有在没有异常时才会执行的代码
- `finally` 无论是否有异常，都会执行的代码

```
try:
    # 提示用户输入一个整数
    num = int(input('输入一个整数：'))
    # 使用 8 除以用户输入的整数并输出
    result = 8 / num
    print(result)
except ValueError:
    print('请输入正确的整数')
except Exception as e:
    print('未知错误 %s' % e)
else:
    print('尝试成功')
finally:
    print('无论是否出现错误，都会执行的代码')

print('-' * 50)
```

---

069. 异常-06-异常的传递性

- **异常的传递** —— 当 **函数/方法** 执行 **出现异常**，会 **将异常传递** 给函数/方法的 **调用一方**
- 如果 **传递到主程序**，仍然 **没有异常处理**，**程序才会被终止**

提示

- 在开发中，可以在主函数中增加 **异常捕获**
- 而在主函数中调用的其他函数，只要出现异常，都会传递到主函数的 **异常捕获** 中
- 这样就不需要在代码中，增加大量的 **异常捕获**，能够保证代码的整洁

需求

1. 定义函数 `demo1()` **提示用户输入一个整数并且返回**
2. 定义函数 `demo2()` 调用 `demo1()`
3. 在主程序中调用 `demo2()`

示例程序

```
def demo1():
    return int(input('输入整数：'))


print(demo1())
```

输入 `a` 之后的错误信息

```
输入整数：a
Traceback (most recent call last):
  File "D:/code/python-fullstack/darkhorse/10_异常/hm_05_异常的传递.py", lin
e 5, in <module>
    print(demo1())
  File "D:/code/python-fullstack/darkhorse/10_异常/hm_05_异常的传递.py", lin
e 2, in demo1
    return int(input('输入整数：'))
ValueError: invalid literal for int() with base 10: 'a'
```

上面是在执行子程序 `demo1` 的第 `2` 行时出现了异常，抛到了主程序的第 `5` 行（主程序就是 `demo1` 的调用者），主程序没有处理异常，所以结束运行。

最终代码：

```
def demo1():
    return int(input('输入整数：'))


def demo2():
    return demo1()
```

```python
# 利用异常的传递性，在主程序捕获异常
try:
    print(demo2())
except Exception as e:
    print('未知错误 %s' % e)
```

---

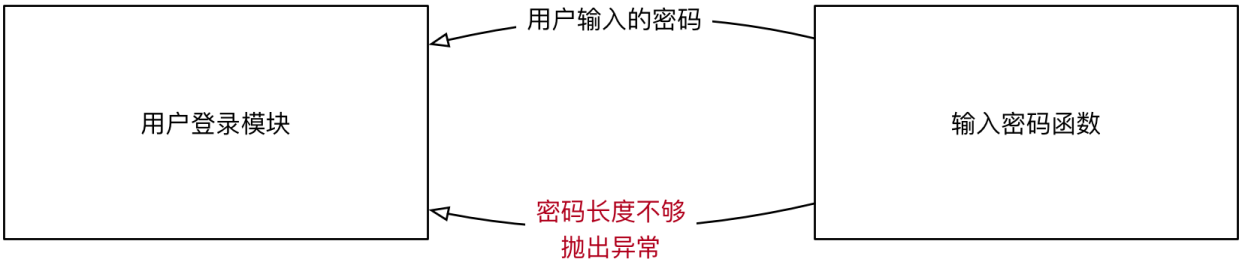
070. 异常-07-主动抛出异常的应用场景

应用场景

- 在开发中，除了代码执行出错 `Python` 解释器会抛出异常之外
- 还可以根据应用程序特有的业务需求主动抛出异常

示例

- 提示用户输入密码，如果长度少于 `8`，抛出异常



注意

- 当前函数 **只负责** 提示用户输入密码，如果 **密码长度不正确，需要其他的函数进行额外处理**
- 因此可以 **抛出异常**，由其他需要处理的函数 **捕获异常**

---

071. 异常-08-主动抛出异常案例演练

- `Python` 中提供了一个 `Exception` **异常类**
- 在开发时，如果满足特定业务需求时，希望 **抛出异常**，可以：
    - 创建 一个 `Exception` 的 **对象**
    - 使用 `raise` 关键字 **抛出异常对象**

需求

- 定义 `input_password` 函数，提示用户输入密码
- 如果用户输入长度 < 8，抛出异常
- 如果用户输入长度 >=8，返回输入的密码

```python
def input_password():
    # 1. 提示用户输入密码
    pwd = input('请输入密码：')
    # 2. 判断密码长度 >= 8，返回用户输入的密码
    if len(pwd) >= 8:
        return pwd
    # 3. 如果 <8 主动抛出异常
    print('主动抛出异常')
    # 1> 创建异常对象 - 可以使用错误信息字符串作为参数
    ex = Exception('密码长度不够')
    # 2> 主动抛出异常
    raise ex


# 提示用户输入密码
try:
    print(input_password())
except Exception as result:
    print(result)
```

072. 模块-01-基本概念和import导入复习

> 模块是 `Python` 程序架构的一个核心概念

- 每一个以扩展名 `py` 结尾的 `Python` 源代码文件都是一个 **模块**
- **模块名** 同样也是一个 **标识符**，需要符合标识符的命名规则
- 在模块中定义的 **全局变量**、**函数**、**类** 都是提供给外界直接使用的 工具
- 模块就好比是 **工具包**，要想使用这个工具包中的工具，就需要先 **导入** 这个模块

`import` 导入

```python
import 模块名1, 模块名2
```

> 提示：在导入模块时，**每个导入应该独占一行**

```python
import 模块名1
import 模块名2
```

导入之后

- 通过 `模块名.` 使用模块提供的工具 —— **全局变量、函数、类**

---

073. 模块-02-import导入时指定别名

使用 `as` 指定模块的别名

> 如果模块的名字太长，可以使用 `as` 指定模块的名称，以方便在代码中的使用

```
import 模块名1 as 模块别名
```

> 注意：**模块别名** 应该符合 **大驼峰命名法**

```
import hm_01_测试模块1 as DogModule
import hm_02_测试模块2 as CatModule

DogModule.say_hello()
CatModule.say_hello()

dog = DogModule.Dog()
print(dog)

cat = CatModule.Cat()
print(cat)
```

输出结果

```
我是 模块1
我是 模块2
<hm_01_测试模块1.Dog object at 0x000001D73154DEF0>
<hm_02_测试模块2.Cat object at 0x000001D73154DF60>
```

控制台输出的模块名还是原来的，别名只是方便我们编码的时候使用，并不会更改真正的模块名。

> **【!】此处存疑**：模块名应该全部小写，而不是用大驼峰法命名！

---

074. 模块-03-from import局部导入

- 如果希望从某一个模块 中，导入 **部分** 工具，就可以使用 `from ... import` 的方式
- `import 模块名` 是 **一次性** 把模块中所有工具全部导入，并且通过 **模块名/别名** 访问

```
# 从 模块 导入 某一个工具
from 模块名1 import 工具名
```

导入之后

- 不需要通过 `模块名.`
- 可以直接使用模块提供的工具 —— **全局变量、函数、类**

```
from hm_01_测试模块1 import Dog
from hm_02_测试模块2 import say_hello

say_hello()
wangcai = Dog()
print(wangcai)
```

---

075. 模块-04-from import导入同名工具

注意

> 如果 **两个模块**，存在 **同名的函数**，那么 **后导入模块的函数，会覆盖掉先导入的函数**

- 开发时 `import` 代码应该统一写在 **代码的顶部**，更容易及时发现冲突
- 一旦发现冲突，可以使用 `as` 关键字 **给其中一个工具起一个别名**

```
# from hm_01_测试模块1 import say_hello
from hm_02_测试模块2 import say_hello as module2_say_hello
from hm_01_测试模块1 import say_hello

say_hello()
module2_say_hello()
```

---

076. 模块-05-from import导入所有工具

`from ... import *`

```
# 从 模块 导入 所有工具
from 模块名1 import *
```

示例：

```
from hm_01_测试模块1 import *

print(title)
say_hello()

wangcai = Dog()
print(wangcai)
```

注意

- 这种方式不推荐使用，因为函数重名并没有任何的提示，出现问题不好排查

---

077. 模块-06-模块搜索顺序

`Python` 的解释器在 **导入模块** 时，会：

1. 搜索 **当前目录** 指定模块名的文件，**如果有就直接导入**
2. 如果没有，再搜索 **系统目录**

> 在开发时，给文件起名，**不要和系统的模块文件重名**

`Python` 中每一个模块都有一个内置属性 `__file__` 可以 **查看模块的完整路径**

示例：

```
import random

# 生成一个 0～10 的数字
rand = random.randint(0, 10)

print(rand)
```

> 注意：如果当前目录下，存在一个 `random.py` 的文件，程序就无法正常执行了！

会报错

```
Traceback (most recent call last):
  File "D:/code/python-fullstack/darkhorse/11_模块/hm_08_模块的搜索顺序.py",
 line 3, in <module>
    rand = random.randint(0, 10)
AttributeError: module 'random' has no attribute 'randint'
```

- 这个时候，`Python` 的解释器会 **加载当前目录** 下的 `random.py` 而不会加载 **系统的** `random` 模块

使用下面的代码查看模块路径 `__file__`

```
import random
print(random.__file__)

# rand = random.randint(0, 10)

# print(rand)
```

得到输出

```
D:\code\python-fullstack\darkhorse\11_模块\random.py
```

现在删除工作目录的 `random.py`，得到输出

```
C:\Users\jpch89\AppData\Local\Programs\Python\Python36\lib\random.py
```

---

078. 模块-07-开发原则以及导入文件时会执行没有缩进的代码

每一个文件都应该是可以被导入的

- 一个独立的 `Python` 文件就是一个模块
- 在导入文件时，文件中**所有没有任何缩进的代码都会被执行一遍**！

实际开发场景

- 在实际开发中，每一个模块都是独立开发的，大多都有专人负责

- **开发人员** 通常会在 **模块下方增加一些测试代码**
- 这些测试代码仅在模块内使用，而被导入到其他文件中不需要执行

---

079. 模块-08-`__name__`属性兼顾测试和导入两种模式

`__name__` 属性

- `__name__` 属性可以做到，测试模块的代码只在测试情况下被运行，而在被导入时不会被执行！
- `__name__` 是 `Python` 的一个内置属性，记录着一个 **字符串**
- 如果 **是被其他文件导入的**，`__name__` 就是 `"模块名"`
- 如果 **是当前执行的程序**，`__name__` 是 `"__main__"`

在很多 `Python` 文件中都会看到以下格式的代码：

```
# 导入模块
# 定义全局变量
# 定义类
# 定义函数

# 在代码的最下方
def main():
    # ...
    pass

# 根据 __name__ 判断是否执行下方代码
if __name__ == "__main__":
    main()
```

---

080. 包-01-包的概念以及建立包的方式

概念

- **包** `package` 是一个**包含多个模块的特殊目录**
- 目录下有一个**特殊的文件** `__init__.py`
- 包名的**命名方式和变量名一致**，小写字母、数字和 `_`，不能以数字开头

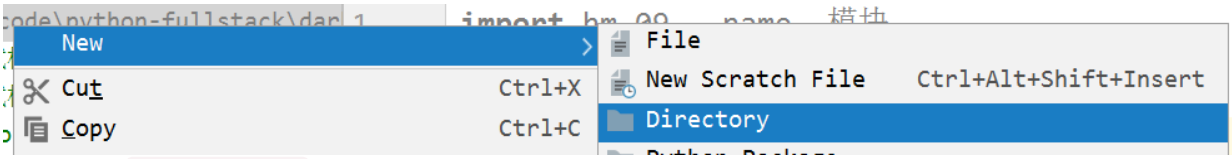
好处

- 使用 `import 包名` 可以一次性导入包中所有的模块

案例演练

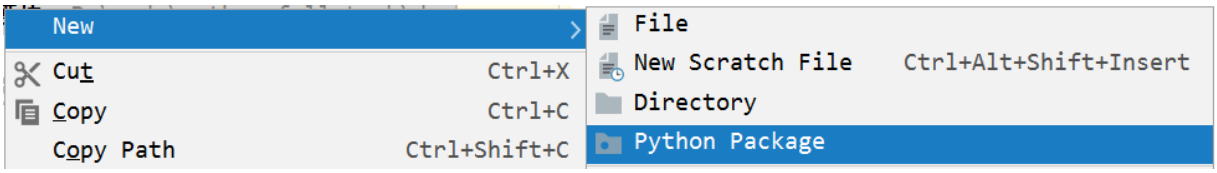
- 新建一个 `hm_message` 的 包
- 在目录下，新建两个文件 `send_message` 和 `receive_message`
- 在 `send_message` 文件中定义一个 `send` 函数
- 在 `receive_message` 文件中定义一个 `receive` 函数
- 在外部直接导入 `hm_message` 的包

使用目录建立包



然后新建 `__init__.py` 文件

直接新建包



---

081. 包-02-封装模块、设置`__init__`和外界导入包

`__init__.py`

- 要在外界使用包中的模块，需要在 `__init__.py` 中**指定对外界提供的模块列表**

```
# 从 当前目录 导入 模块列表
from . import send_message
from . import receive_message
```

---

082. 制作模块-01-明确目的和介绍步骤

如果希望自己开发的模块，**分享**给其他人，可以按照以下步骤操作。

制作发布压缩包步骤

1. 创建 `setup.py`

```
from distutils.core import setup
```

```
setup(name="hm_message",  # 包名
      version="1.0",  # 版本
      description="jpch89's 发送和接收消息模块",  # 描述信息
      long_description="完整的发送和接收消息模块",  # 完整描述信息
      author="jpch89",  # 作者
      author_email="jpch89@outlook.com",  # 作者邮箱
      url="www.jpch89.com",  # 主页
      py_modules=["hm_message.send_message",
                  "hm_message.receive_message"])
```

有关字典参数的详细信息，可以参阅官方网站：
https://docs.python.org/2/distutils/apiref.html

2. 构建模块

```
$ python3 setup.py build
```

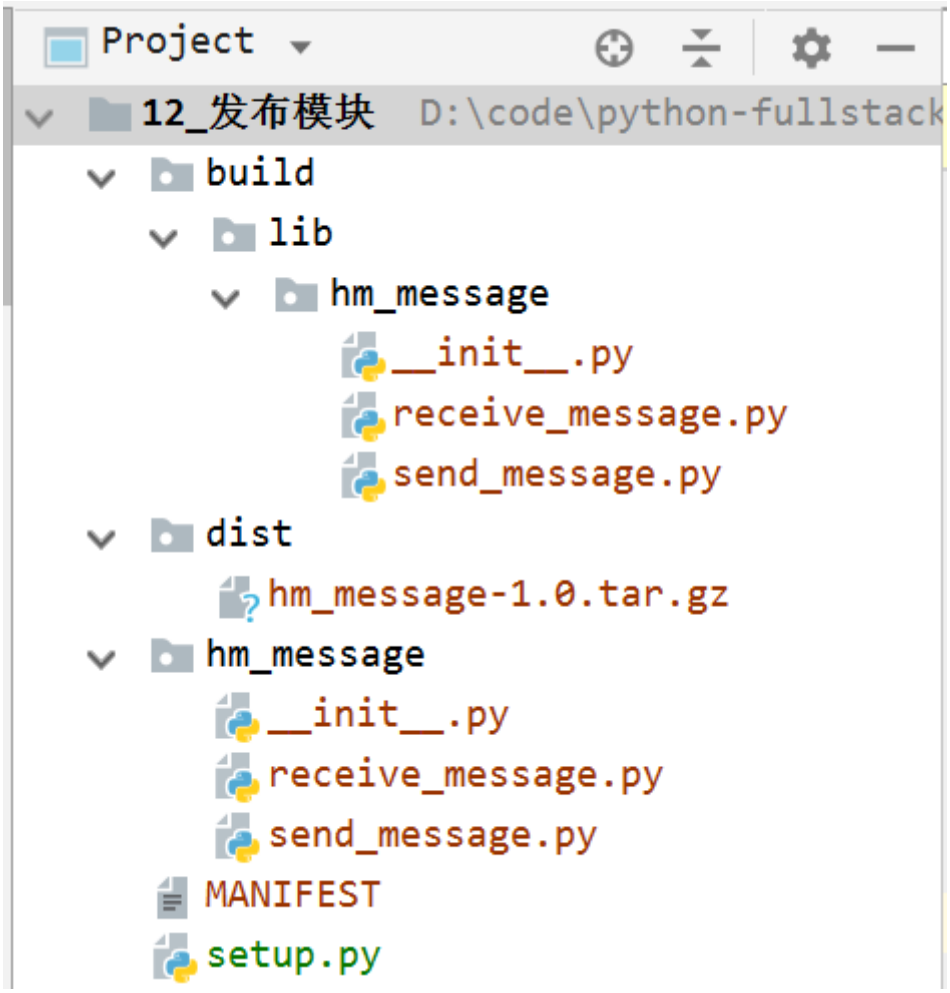
3. 生成发布压缩包

```
$ python3 setup.py sdist
```

> **注意**：要制作哪个版本的模块，就使用哪个版本的解释器执行！

---

083. 制作模块-02-制作模块压缩包



084. 制作模块-03-安装模块压缩包

安装模块

```
$ tar -zxvf hm_message-1.0.tar.gz
$ sudo python3 setup.py install
```

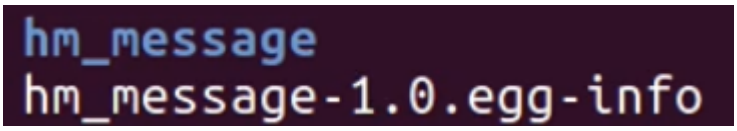
085. 制作模块-04-卸载已经安装过的模块

如何查看模块/包的完整路径

- 先导入
- `print(包名.__file__)`

卸载模块

- 删除一个目录和一个文件



- 直接从安装目录下，把安装模块的目录删除就可以

```
$ cd /usr/local/lib/python3.5/dist-packages/
$ sudo rm -r hm_message*
```

---

086. pip-使用pip安装pygame模块

- **第三方模块** 通常是指由 **知名的第三方团队** 开发的并且 **被程序员广泛使用** 的 `Python` 包 /模块
- 例如 `pygame` 就是一套非常成熟的 **游戏开发模块**
- `pip` 是一个现代的，通用的 `Python` **包管理工具**
- 提供了对 `Python` 包的查找、下载、安装、卸载等功能

安装和卸载命令如下：

```
# 将模块安装到 Python 2.x 环境
$ sudo pip install pygame
$ sudo pip uninstall pygame

# 将模块安装到 Python 3.x 环境
$ sudo pip3 install pygame
$ sudo pip3 uninstall pygame
```

在 `Mac` 下安装 `iPython`

```
$ sudo pip install ipython
```

在 `Linux` 下安装 `iPython`

```
$ sudo apt install ipython
$ sudo apt install ipython3
```

---

087. 文件-文件概念以及文本文件和二进制文件的区别

文件的概念

- 计算机的 **文件**，就是存储在某种 **长期储存设备** 上的一段 数据
- 长期存储设备包括：硬盘、U 盘、移动硬盘、光盘…

文件的作用

- 将数据长期保存下来，在需要的时候使用

文件的存储方式

- 在计算机中，文件是以 **二进制** 的方式保存在磁盘上的

文本文件和二进制文件
文本文件

- 可以使用 **文本编辑软件** 查看
- 本质上还是二进制文件
- 例如：`python` 的源程序

二进制文件

- 保存的内容不是给人直接阅读的，而是 **提供给其他软件使用的**
- 例如：图片文件、音频文件、视频文件等等
- 二进制文件不能使用 **文本编辑软件** 查看

---

088. 文件操作-01-文件操作套路以及Python中的对应函数和方法

在计算机中要操作文件的套路非常固定，一共包含三个步骤：

1. 打开文件
2. 读、写文件
    - **读** 将文件内容读入内存
    - **写** 将内存内容写入文件
3. 关闭文件

在 `Python` 中要操作文件需要记住 `1` **个函数和** `3` **个方法**

序号	函数/方法	说明
01	`open`	打开文件，并且返回文件操作对象

02	`read`	将文件内容读取到内存
03	`write`	将指定内容写入文件
04	`close`	关闭文件

- `open` 函数负责打开文件，并且返回文件对象
- `read/write/close` 三个方法都需要通过 **文件对象** 来调用

---

089. 文件操作-02-读取文件内容

- `open` 函数的第一个参数是要打开的文件名（文件名区分大小写）
  - 如果文件存在，**返回文件操作对象**
  - 如果文件不存在，**抛出异常**

- `read` 方法可以**一次性读入并返回文件的所有内容**
- `close` 方法负责**关闭文件**
  如果忘记关闭文件，会造成系统资源消耗，而且会影响到后续对文件的访问
- 注意：`read` 方法执行后，会把 **文件指针** 移动到 **文件的末尾**

```
# 1. 打开文件
file = open('README', encoding='utf-8')

# 2. 读取文件内容
text = file.read()
print(text)

# 3. 关闭文件
file.close()
```

> 【!】如果不加入 `encoding` 会有乱码！

提示

- 在开发中，通常会先编写打开和关闭的代码，再编写中间针对文件的读/写操作！

---

090. 文件操作-03-读取文件后文件指针会发生变化

文件指针

- **文件指针** 标记 **从哪个位置开始读取数据**
- **第一次打开** 文件时，通常 **文件指针会指向文件的开始位置**
- 当执行了 `read` 方法后，文件指针 **会移动到 读取内容的末尾**
- 默认情况下会移动到 **文件末尾**

思考

- 如果执行了一次 `read` 方法，读取了所有内容，那么**再次**调用 `read` 方法，还能够获得到内容吗？

答案

- **不能**
- 第一次读取之后，文件指针移动到了文件末尾，再次调用不会读取到任何的内容

```
# 1. 打开文件
file = open('README', encoding='utf-8')

# 2. 读取文件内容
text = file.read()
print(text)
print(len(text))
print('-' * 50)
text = file.read()
print(text)
print(len(text))

# 3. 关闭文件
file.close()
```

---

091. 文件操作-04-打开文件方式以及写入和追加数据

`open` 函数默认以**只读方式**打开文件，并且**返回文件对象**

> 【!】默认应该是 `rt` 模式打开，即只读文本模式

语法如下：

```
f = open("文件名", "访问方式")
```

访问方式	说明
`r`	以只读方式打开文件。文件的指针将会放在文件的开头，这是默认模式。如果文件不存在，抛出异常
`w`	以只写方式打开文件。如果文件存在会被覆盖。如果文件不存在，创建新文件
`a`	以追加方式打开文件。如果该文件已存在，文件指针将会放在文件的结尾。如果文件不存在，创建新文件进行写入
`r+`	以读写方式打开文件。文件的指针将会放在文件的开头。如果文件不存在，抛出异常
`w+`	以读写方式打开文件。如果文件存在会被覆盖。如果文件不存在，创建新文件
`a+`	以读写方式打开文件。如果该文件已存在，文件指针将会放在文件的结尾。如果文件不存在，创建新文件进行写入

提示

- 频繁的移动文件指针，会影响文件的读写效率，开发中更多的时候会以 **只读、只写** 的方式来操作文件

---

092. 文件操作-05-使用readline分行读取大文件

- `read` 方法默认会把文件的 **所有内容** 一次性读取到内存
- 如果文件太大，对内存的占用会非常严重

`readline` 方法

- `readline` 方法可以一次读取一行内容
- 方法执行后，会把 **文件指针** 移动到下一行，准备再次读取

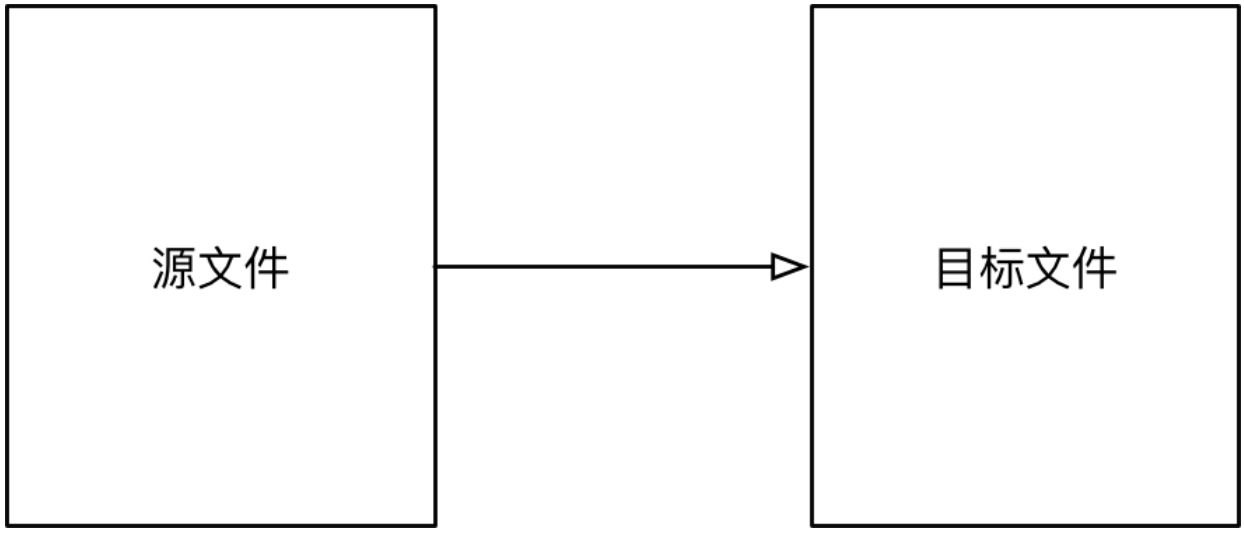
读取大文件的正确姿势

```
# 打开文件
file = open("README")

while True:
    # 读取一行内容
    text = file.readline()

    # 判断是否读到内容
    if not text:
        break
```

```
    # 每读取一行的末尾已经有了一个 `\n`
    print(text, end="")

# 关闭文件
file.close()
```

093. 文件操作-06-小文件复制

目标
用代码的方式，来实现文件复制过程



小文件复制

- 打开一个已有文件，读取完整内容，并写入到另外一个文件

```
# 1. 打开文件
file_read = open("README")
file_write = open("README[复件]", "w")

# 2. 读取并写入文件
text = file_read.read()
file_write.write(text)

# 3. 关闭文件
file_read.close()
file_write.close()
```

094. 文件操作-07-大文件复制

大文件复制

- 打开一个已有文件，逐行读取内容，并顺序写入到另外一个文件

```
# 1. 打开文件
file_read = open("README")
file_write = open("README[复件]", "w")

# 2. 读取并写入文件
while True:
    # 每次读取一行
    text = file_read.readline()

    # 判断是否读取到内容
    if not text:
        break

    file_write.write(text)

# 3. 关闭文件
file_read.close()
file_write.close()
```

---

095. 导入os模块，执行文件和目录管理操作

- 在 **终端/文件浏览器** 中可以执行常规的 **文件/目录管理操作**，例如：创建、重命名、删除、改变路径、查看目录内容、……
- 在 `Python` 中，如果希望通过程序实现上述功能，需要导入 `os` 模块
  `import os`

文件操作

序号	方法名	说明	示例
01	`rename`	重命名文件	`os.rename(源文件名, 目标文件名)`
02	`remove`	删除文件	`os.remove(文件名)`

目录操作

序号	方法名	说明	示例
01	`listdir`	目录列表	`os.listdir(目录名)`

02	`mkdir`	创建目录	`os.mkdir(目录名)`
03	`rmdir`	删除目录	`os.rmdir(目录名)`
04	`getcwd`	获取当前目录	`os.getcwd()`
05	`chdir`	修改工作目录	`os.chdir(目标目录)`
06	`path.isdir`	判断是否是文件	`os.path.isdir(文件路径)`

> 提示：文件或者目录操作都支持 **相对路径** 和 **绝对路径**

---

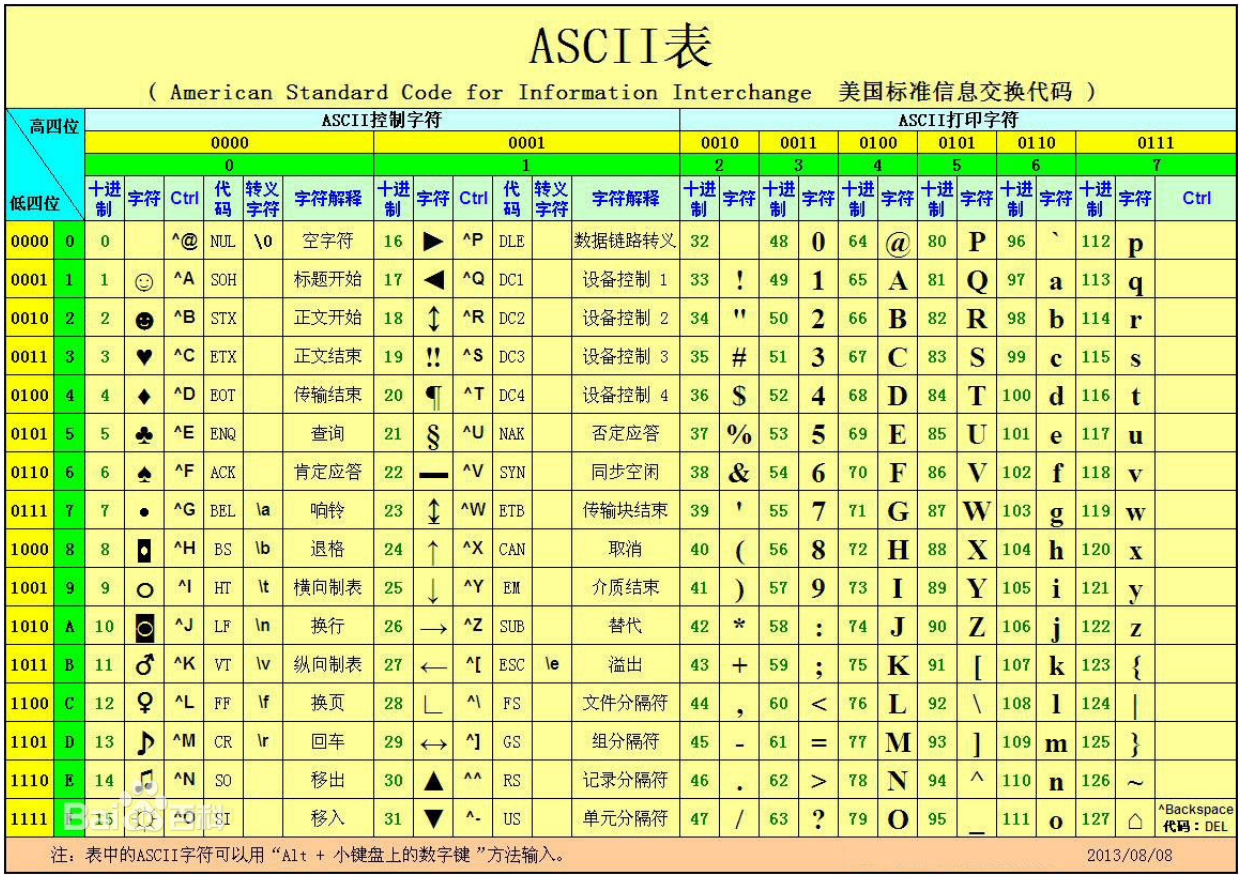
096. 文本编码-01-文本文件的编码方式 ASCII和UTF8

文本文件存储的内容是基于 **字符编码** 的文件，常见的编码有 `ASCII` 编码，`UNICODE` 编码等

- `Python 2.x` 默认使用 `ASCII` 编码格式
- `Python 3.x` 默认使用 `UTF-8` 编码格式

`ASCII` 编码

- 计算机中只有 `256` 个 `ASCII` 字符
- 一个 `ASCII` 在内存中占用 `1` **个字节** 的空间
- `8` 个 `0/1` 的排列组合方式一共有 `256` 种，也就是 `2 ** 8`



ASCII表

(American Standard Code for Information Interchange 美国标准信息交换代码)

高四位 / 低四位		ASCII控制字符												ASCII打印字符																	
		0000						0001						0010		0011		0100		0101		0110		0111							
		0						1						2		3		4		5		6		7							
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl			
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p						
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q						
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r						
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s						
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t						
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u						
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v						
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w						
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x						
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y						
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z						
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{						
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|					
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}						
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~						
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL					

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。 2013/08/08

`UTF-8` 编码格式

- 计算机中使用 `1~6` 个字节 来表示一个 `UTF-8` 字符，涵盖了 **地球上几乎所有地区的文字**
- 大多数汉字会使用 `3` **个字节** 表示
- `UTF-8` 是 `UNICODE` 编码的一种编码格式

---

097. 文本编码-02-怎么样在Python2.x中使用中文

- `Python 2.x` 默认使用 `ASCII` 编码格式
- `Python 3.x` 默认使用 `UTF-8` 编码格式

在 `Python 2.x` 文件的 **第一行** 增加以下代码，解释器会以 `utf-8` 编码来处理 `python` 文件

```
# *-* coding:utf8 *-*
```

> 这方式是**官方推荐**使用的！

也可以使用

```
# coding=utf8
```

> 【!】上面写的**有误**
> 官方规定在 `PEP 263 -- Defining Python Source Code Encodings`
> https://www.python.org/dev/peps/pep-0263/
> 这个叫做**魔法注释** `magic comment`
> 位于文件的**第一行**或者**第二行**
> 要满足以下正则表达式，第一个组就是本文件的编码格式
> `^[ \t\f]*#.*?coding[:=][ \t]*([-_.a-zA-Z0-9]+)`
> 前两行这么写比较好

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

---

098. 文本编码-03-Python2.x处理中文字符串

`unicode` 字符串

- 在 `Python 2.x` 中，即使指定了文件使用 `UTF-8` 的编码格式，但是在遍历字符串时，仍然会 **以字节为单位遍历** 字符串
- 要能够 **正确的遍历字符串**，在定义字符串时，需要 **在字符串的引号前**，增加一个小写字母 `u`，告诉解释器这是一个 `unicode` 字符串（使用 `UTF-8` 编码格式的字符串）

```
# -*- coding: utf-8 -*-

# 在字符串前，增加一个 `u` 表示这个字符串是一个 utf8 字符串
hello_str = u"你好世界"

print(hello_str)

for c in hello_str:
    print(c)
```

---

099. eval-01-基本使用

`eval` 函数

- `eval()` 函数十分强大 —— **将字符串当成有效的表达式来求值并返回计算结果**

```
# 基本的数学计算
In [1]: eval("1 + 1")
Out[1]: 2

# 字符串重复
In [2]: eval("'*' * 10")
Out[2]: '**********'

# 将字符串转换成列表
In [3]: type(eval("[1, 2, 3, 4, 5]"))
Out[3]: list

# 将字符串转换成字典
In [4]: type(eval("{'name': 'xiaoming', 'age': 18}"))
Out[4]: dict
```

案例：计算器

```
input_str = input('请输入算术题：')
print(eval(input_str))
```

---

100. eval-02-[扩展]不要直接转换input结果

不要滥用 `eval`

- 在开发时千万不要使用 `eval` 直接转换 `input` 的结果

```
__import__('os').system('ls')
```

等价代码

```
import os

os.system("终端命令")
```

- 执行成功，返回 `0`
- 执行失败，返回错误信息

完成于 `201810091222`