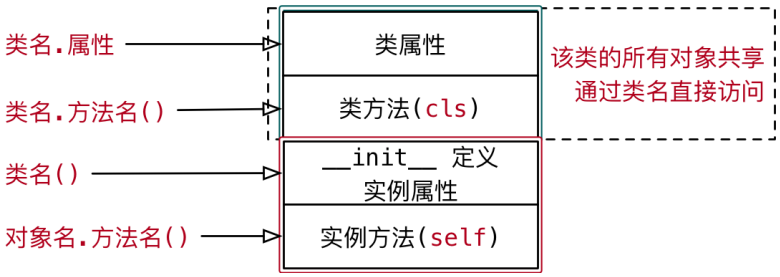
Day-7

- Video-047-057-类属性 & 类方法
  - 类属性——类是一个特殊的对象
    - 类在程序运行时，同样会被加载到内存
    - 通过**类名.**的方式可以访问类的属性 或者 调用类的方法



    - 类属性：记录与这个类相关的特征
      - 通过赋值语句，定义类属性
      - 属性获取机制：**向上查找机制**，首先在对象内部查找对象属性，没有找到就会向上寻找类属性
      - 访问类属性方法：**类名.属性名**
  - 类方法——针对类对象定义的方法
    - 语法

```
@classmethod
def 类方法名(cls):
    pass
```

- 类方法需要用 **修饰器** `@classmethod` 来标识，**告诉解释器这是一个类方法**
- 类方法的 **第一个参数** 应该是 `cls`
  - 由 **哪一个类** 调用的方法，方法内的 `cls` 就是 **哪一个类的引用**
  - 这个参数和 **实例方法** 的第一个参数是 `self` 类似
  - **提示** 使用其他名称也可以，不过习惯使用 `cls`

3. 通过 **类名.** 调用 **类方法，调用方法时，** 不需要传递 `cls` 参数
4. **在方法内部**
   - 可以通过 `cls.` **访问类的属性**
   - 也可以通过 `cls.` **调用其他的类方法**

    - 1.@classmethod 修饰器，告诉解释器这是一个类方法
    - 2.第一个参数是cls，由哪一个类调用的方法，方法内的cls就是哪一个类的引用
    - 3.方法内部
      - 1.cls.访问类的属性
      - 2.cls.调用其他的类方法
    - 静态方法

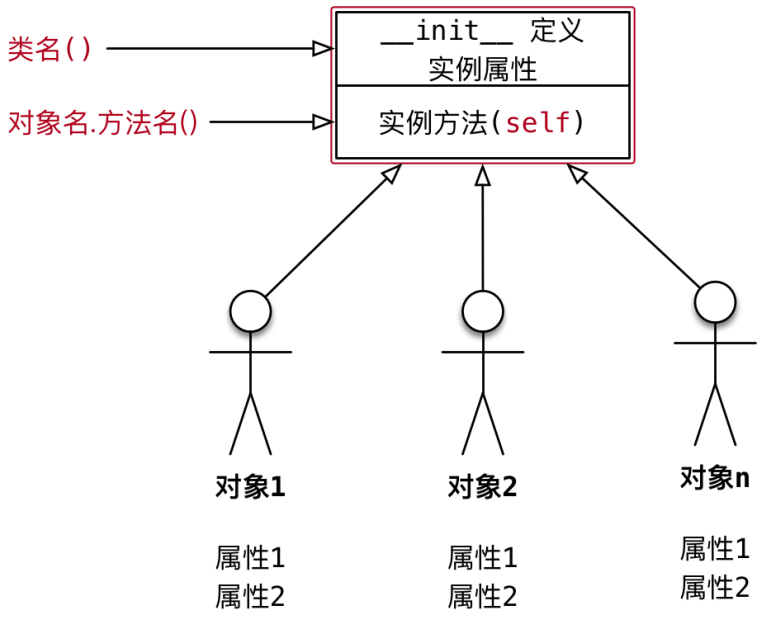
- 既不需要访问实例属性或者调用实例方法，也不需要访问类属性或者调用类方法
- 语法

```
@staticmethod
def 静态方法名():
    pass
```

  - **静态方法** 需要用 **修饰器** `@staticmethod` 来标识，**告诉解释器这是一个静态方法**
  - 通过 **类名.** 调用 **静态方法**

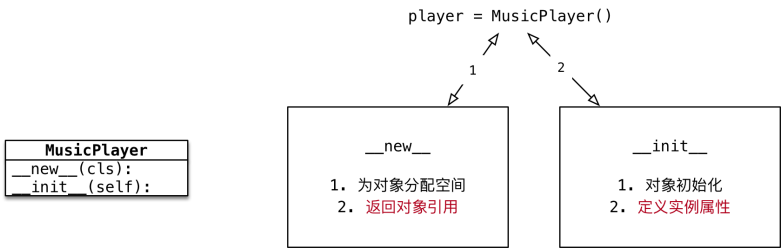
- 总结
  - **实例方法** —— 方法内部需要访问 **实例属性**
    - **实例方法** 内部可以使用 **类名.** 访问类属性
  - **类方法** —— 方法内部 **只** 需要访问 **类属性**
  - **静态方法** —— 方法内部，不需要访问 **实例属性** 和 **类属性**
  - 既要访问实例属性，有需要访问类属性，则定义为实例方法

- **实例：创建出来的对象叫做类的实例**
  - 1.使用面相对象开发，第一步是设计类
  - 2.使用类名() 创建对象，创建对象的动作有两布
    - 调用初始化方法 __init__ 为 **对象初始化**
    - 在内存中为对象 **分配空间**
  - 3.对象创建后，内存中就有类一个对象的实实在在的存在——实例

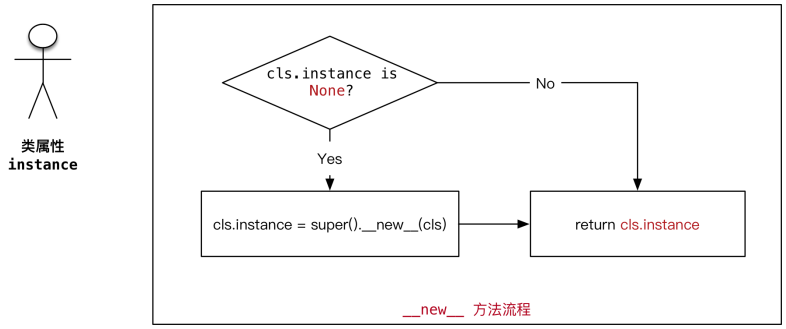


  - 类的方法只有一份
  - 程序执行时
    - 1.对象各自拥有自己的实例属性，属性拥有独立的内存空间，保存各自不同的属性
    - 2.调用对象方法，可以通过self，方法在内存中只有一份，调用时需要把对象的引用传递到方法内部

- Video-058-064-单例
  - 设计模式：前人工作的总结和提炼
    - 通常，被人们广泛流传的设计模式都是针对某一特定问题的成熟的解决方案
    - 使用设计模式是为了可重用代码，让代码更容易被他人理解、保证代码可靠性
  - 单例设计模式
    - **目的** —— 让 **类** 创建的对象，在系统中 **只有 唯一的一个实例**
    - 每一次执行 类名() 返回的对象，**内存地址是相同的**
  - __new__ 方法：内置的静态方法
    - 作用
      - 在内存中为对象 **分配空间**
      - **返回** 对象的引用
      - **Python解释器获得对象的引用后，将引用作为第一个参数，传递给__init__方法**
  - __new__ 是一个**静态方法**，让类创建对象时，仅有一个实例，实现单例设计模式



    - 一定要有返回值：return super().__new__(cls)
      - **否则 Python 的解释器 得不到 分配了空间的 对象引用，就不会调用对象的初始化方法**
  - 单例——让类创建的对象，在系统中只有唯一的一个实例
    - 定义一个 **类属性**，初始值是 None，用于记录 **单例对象的引用**
    - 重写 __new__ 方法
    - 如果 **类属性** is None，调用父类方法分配空间，并在类属性中记录结果
    - 返回 **类属性** 中记录的 **对象引用**



  - 只执行一次初始化方法
    - 定义一个类属性 init_flag 标记是否 **执行过初始化动作**，初始值为 False
    - 在 __init__ 方法中，判断 init_flag，如果为 False 就执行初始化动作

    - 然后将 init_flag 设置为 True
    - 这样，再次 **自动** 调用 __init__ 方法时，**初始化动作就不会被再次执行** 了
- Video-065-071异常
    - 概念：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**

    - 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**
    - 捕获异常
        - 如果 **对某些代码的执行不能确定是否正确**，可以增加 try(尝试) 来 **捕获异常**
        - y语法

```
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**，下方编写尝试失败的代码
        - 错误类型捕获
            - 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**
            - 语法

```
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**
        - 捕获未知错误

        - **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

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

    - 异常捕获的完整语法

```
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
    - 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
        - 只需要利用异常的传递性，在主程序捕获异常即可
    - 抛出异常，raise
        - **根据应用程序特有的业务需求主动抛出异常**
        - Exception类，创建异常对象，使用raise关键字抛出异常对象
- Video-072-086模块
    - 概念——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
    - **导入：import导入，可以使用as指定模块的别名**
        - 模块别名应该符合大驼峰命名法
    - 通过 模块名. 使用 **模块提供的工具** —— **全局变量**、**函数**、**类**

- from ... import ... 导入部分工具
  - **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**
  - 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**
    - 使用as关键字，起别名来区分
  - from ... import * 导入所有工具
- 模块的搜索顺序
  - Python解释器在导入模块时
    - 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**
- 每一个文件都应该是可以被导入都
  - 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！
- __name__ 属性
  - **作用：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__`
  - 开发模式

    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()
    ```

- 包：包含多个模块的特殊目录
  - 包含特殊文件__init__.py
  - 包名的 **命名方式** 和变量名一致，**小写字母** + _
  - 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

- 发布模块Video-082-085
  - 制作模块压缩包
    - 1.创建setup.py文件

```
from distutils.core import setup

setup(name="hm_message",  # 包名
      version="1.0",  # 版本
      description="itheima'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="itheima",  # 作者
      author_email="itheima@itheima.com",  # 作者邮箱
      url="www.itheima.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
  - 安装模块
- Video-087-098文件
  - 文件的概念：**存储在某种长期存储设备上的一段数据**
  - 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
  - 文件类型，**本质上都是二进制存储**
    - 文本文件：可以使用文本编辑软件查看
    - 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，不能使用文本编辑软件查看
  - 基本操作
    - 套路
      - 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
    - 操作文件的函数和方法
      - open函数，打开文件，并且**返回文件操作对象**
        - 打开方式，默认是只读方式，并且返回文件对象

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

- 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

- 方法
  - read方法：将文件内容读取到内存
    - 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**，会 **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**
  - write方法：将指定内存写入文件
  - close方法：关闭文件
  - readline方法：按行读取文件内容
    - **方法执行后，会把文件指针移动到下一行，准备再次读取**
  - 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！
  - 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**
  - **三个方法都需要通过文件对象来调用**

- 文件/目录的常用管理操作，os模块
  - 创建、重命名、删除、改变路径、查看目录和内容....
  - 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**

- 文本文件的编码格式
  - 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
  - UTF-8编码
    - 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
- Video-099-100eval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**
  - 不要滥用eval

以上内容整理于 幕布文档