人工智能与
Python程序设计

回顾

- 1. 文件概述
- 2. 文件读写
- 3. 多维数据的格式化与处理
- 4. os模块编程

文件概述

- 文件是存储在外部介质（存储器）上的数据序列，可以包含多种数据内容，例如程序文件、数字图像、音频、视频等。

- 文件包括两种类型：文本文件和二进制文件

- 文件路径
  - 计算机文件系统中，用于描述文件或文件夹在文件系统中存储位置的字符串
  - 文件路径通常包括用特定符号分割的目录名称（文件夹名称）和文件名
    - Windows: C:\ProgramData\Anaconda3\Scripts\conda.exe
    - Mac: /Users/defaultstr/anaconda3/bin/conda
  - 允许用户在文件系统中快速定位文件并对其进行操作，例如打开、复制、移动或删除。

文件读写

- 文件操作
  - 文件读写是最常见的IO操作，遵循打开-操作-关闭步骤。
  - Python内置了读写文件的函数，请求操作系统打开一个文件对象
- 文件的打开
  - 利用Python内置的open()函数，以读文件的模式打开一个文件对象
  - <name>：文件路径及名称
  - <mode>：打开模式
  - <variable>：文件对象名

```
<variable> = open(<name>, <mode>)
```

文件打开模式	含义
r	只读。如果文件不存在，则返回异常
w	覆盖写。文件不存在则自动创建，存在则覆盖原文件
x	创建写。文件不存在则自动创建；存在则返回异常
a	追加写。文件不存在则自动创建；存在则在原文件最后追加写
b	二进制文件模式
t	文本文件模式（默认）
+	与r/w/x/a一同使用，在原功能基础上增加同时读写功能

文件读写

- 文件的读取
  - 文件打开后，可根据打开方式对文件进行读写操作。
  - 文件读取操作：

操作方法	含义
<variable>. read(size=-1)	根据给定的size参数，读取前size长度的字符串或字节流
<variable>. readline(size=-1)	根据给定的size参数，读取该行前size长度的字符串或字节流
<variable>. readlines(hint=-1)	给定参数，从文件中读入前hint行，并以每行为元素形成一个列表

文件读写

- 文件的写入
  - 从计算机内存向文件写入数据
  - Python提供3个与文件内容写入有关的方法

操作方法	含义
<variable>.write()	向文件写入一个字符串或字节流
<variable>.writelines()	将一个元素全为字符串的列表写入文件
<variable>.seek(offset)	改变当前文件操作指针的位置，offset的值： 0--文件开头；1--当前位置； 2--文件结尾

文件读写

文件的open()与close()操作

```
file_write = open('人工智能与Python程序设计-写入.txt', 'w')
file_write.write('人工智能与Python程序设计 第一课')
file_write.close()
file_read = open('人工智能与Python程序设计-写入.txt', 'r')
print(file_read.read())
file_read.close()
```

```
人工智能与Python程序设计 第一课
```

with...as...操作

```
with open('人工智能与Python程序设计-写入.txt', 'w') as f:
    f.write('人工智能与Python程序设计 第一课')

with open('人工智能与Python程序设计-写入.txt', 'r') as f:
    print(f.read())
```

```
人工智能与Python程序设计 第一课
```

人工智能与
Python程序设计

提纲

- 1. 文件概述
- 2. 文件读写
- 3. 多维数据的格式化与处理
- 4. os模块编程

os模块编程

- 操作系统层级
  - 命令行下，可以通过输入操作系统提供的各种命令实现文件、目录操作
- Python
  - Python内置的os模块可以直接调用操作系统提供的接口函数。
  - os模块提供了多数操作系统的功能接口函数。当os模块被导入后，它会自适应于不同的操作系统平台，根据不同的平台进行相应的操作。

```
import os
os.name
```

```
'posix'
```

如果是posix，说明系统是Linux、Unix或Mac OS X，
如果是nt，就是Windows系统。

os模块编程

- 系统相关信息查询
  - 操作系统的详细信息可以通过os.uname()查看
  - 在操作系统中定义的环境变量，全部保存在os.environ这个变量中。
  - 要获取某个环境变量的值，可以调用os.environ.get('key')

```
os.uname()
```

```
posix.uname_result(sysname='Darwin', nodename='DTaodeMacBook-Pro.local', release='19.6.0', version='Darwin Kernel Version 19.6.0: Mon Aug 31 22:12:52 PDT 2020; root:xnu-6153.141.2~1/RELEASE_X86_64', machine='x86_64')
```

```
os.environ
```

```
environ{'TERM_SESSION_ID': 'w0t0p0:4AB55AA5-2156-4FD4-9539-D89BDD1190E4',
        'SSH_AUTH_SOCK': '/private/tmp/com.apple.launchd.oCFJmavu4X/Listeners',
        'LC_TERMINAL_VERSION': '3.3.12',
        'COLORFGBG': '7;0',
        'ITERM_PROFILE': 'Default',
        'XPC_FLAGS': '0x0',
        'LANG': 'zh_CN.UTF-8',
        'PWD': '/Users/dtao',
        'SHELL': '/bin/zsh',
        'SECURITYSESSIONID': '186a8',
        'TERM_PROGRAM_VERSION': '3.3.12',
        'TERM_PROGRAM': 'iTerm.app',
        'PATH': '/Users/dtao/anaconda3/bin:/Users/dtao/anaconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
        'LC_TERMINAL': 'iTerm2',
        'COLORTERM': 'truecolor',
        'COMMAND_MODE': 'unix2003',
        'TERM': 'xterm-color',
        'HOME': '/Users/dtao',
        'TMPDIR': '/var/folders/_c/sjsxq9ds085dhj5hy35jcfch0000gn/T/',
        'USER': 'dtao',
        'XPC_SERVICE_NAME': '0',
        'LOGNAME': 'dtao',
        'LaunchInstanceID': '53E35E9F-9647-4FB4-8786-CA433FFE028A',
        '__CF_USER_TEXT_ENCODING': '0x0:25:52',
        'ITERM_SESSION_ID': 'w0t0p0:4AB55AA5-2156-4FD4-9539-D89BDD1190E4',
        'SHLVL': '1',
        'OLDPWD': '/Users/dtao',
        'CONDA_EXE': '/Users/dtao/anaconda3/bin/conda',
        '_CE_M': '',
        '_CE_CONDA': '',
        'CONDA_PYTHON_EXE': '/Users/dtao/anaconda3/bin/python',
        'CONDA_SHLVL': '1',
        'CONDA_PREFIX': '/Users/dtao/anaconda3',
        'CONDA_DEFAULT_ENV': 'base',
        'CONDA_PROMPT_MODIFIER': '(base) ',
        '_': '/Users/dtao/anaconda3/bin/jupyter',
```

os模块编程

- 目录操作
  - 路径操作一部分在os.path模块中，一部分在os模块中
  - 查看当前的绝对路径：os.path.abspath()
  - 多个目录路径合并：os.path.join()
    - 在Linux/Unix/Mac下，os.path.join()返回拼接`/`；在windows下返回拼接`\`

```
import os
os.path.abspath('.')
```

```
'/Users/dtao/Documents/course/Python/代码数据/第四周第2次'
```

```
os.path.join('/Users/dtao/Documents/course/Python/代码数据/第四周第2次','PythonAI')
```

```
'/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI'
```

os模块编程

- 目录操作
  - 当前目录遍历，返回一个包含由 *path* 指定目录中条目名称组成的列表：os.listdir(*path='.'*)

```
import os
os.listdir() # 遍历当前目录下的文件，并返回条目名称组成的列表
```

```
['IO编程.ipynb',
 '人工智能与Python程序设计.txt',
 '.DS_Store',
 '编码测试.py',
 '未命名1.ipynb',
 '人工智能与Python程序设计-写入.txt',
 '.ipynb_checkpoints',
 'imagenet-write.csv',
 'imagenet.csv']
```

os模块编程

- 目录操作
  - 创建一个名为 path 的目录，应用以数字表示的权限模式 mode：os.mkdir(*path, mode=0o777, *, dir_fd=None*)
  - 移除（删除）目录 path： os.rmdir (*path, *, dir_fd=None*)

```
os.mkdir(os.path.join('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI','os_module'))
os.listdir('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI')
```

```
['os_module']
```

```
os.rmdir(os.path.join('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI','os_module'))
os.listdir('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI')
```

```
[]
```

os模块编程

- 目录操作
  - 路径拆分，把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名： os.path.split(*path*)
  - 文件扩展名获取： os.path.splitext(*path*)
  - 合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

```
os.path.split('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI')
```

```
('/Users/dtao/Documents/course/Python/代码数据/第四周第2次', 'PythonAI')
```

```
os.path.splitext('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI')
```

```
('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/PythonAI', '')
```

```
os.path.splitext('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/IO编程.ipynb')
```

```
('/Users/dtao/Documents/course/Python/代码数据/第四周第2次/IO编程', '.ipynb')
```

《人工智能与Python程序设计》— 面向对象编程（一）

人工智能与Python程序设计 教研组

Python 面向对象编程（一）

- 面向对象编程（Object Oriented Programming， OOP）是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数

Python 面向对象编程（一）

- 面向对象编程（Object Oriented Programming， OOP）是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数
- 面向过程（Procedure Oriented Programming，POP）的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

Python 面向对象编程（一）

- 面向对象编程（Object Oriented Programming， OOP）是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数
- 面向过程（Procedure Oriented Programming，POP）的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。
- 面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

Python 面向对象编程（一）

- 在Python中，所有数据类型都是对象
- 我们也可以自定义对象。
- 通过定义类（class）的方式自定义数据类型

Python 面向对象编程（一）

- 以＂处理学生成绩＂举例，我们来说明POP 与OOP 在程序流程上的不同之处：

假如有几个学生和他们的考试成绩，在POP中可用一个dict进行表示：

```
std1 = {'name': 'Michael', 'score': 98 }
std2 = {'name': 'Bob', 'score': 81 }
std3 = {'name': 'Kristen', 'score': 93 }
```

如果想处理学生成绩，可通过函数实现，如打印学生的成绩：

```
def print_score(std):
    print('%s: %d' % (std['name'], std['score']))

print_score(std1)
```

```
Michael: 98
```

Python 面向对象编程（一）

- 以＂处理学生成绩＂举例，我们来说明POP 与OOP 在程序流程上的不同之处：

若采用OOP，首先考虑的不是程序的执行流程，而是观察这些学生的共性特点，定义一个Student 类型，其实例（即Student 对象）拥有name和score这两个共有属性（Property）。

```
std1 = {'name': 'Michael', 'score': 98 }
std2 = {'name': 'Bob', 'score': 81 }
std3 = {'name': 'Kristen', 'score': 93 }
```

Python 面向对象编程（一）

若要打印一个学生的成绩，先创建出这个学生对应的对象，然后给对象发送一个打印（print_score）消息，让对象把自己的数据打印出来。

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))
```

Python 面向对象编程（一）

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）：

```
Michael = Student('Michael', 98) # 创建实例对象
Kristen = Student('Kristen', 93) # 创建实例对象

Michael.print_score() # 调用类的方法
Kristen.print_score() # 调用类的方法
```

```
Michael: 98
Kristen: 93
```

人工智能与
Python程序设计

提纲

1. 类和实例
2. 数据封装
3. 访问限制

类和实例

- OOP 的设计思想来源于自然界，因为在自然界中，类(class)和实例(instance)的概念非常自然。
  - 类(class): 用来描述具有相同的属性和方法的对象的集合。比如我们定义的Class--Student，是指学生这个概念。
  - 实例(instance)：创建一个类的实例，类的具体对象。比如一个个具体的Student，Michael和Kristen是两个具体的student。

类和实例

- 类和实例是OOP中最为重要的概念
  - 类是抽象的模板，比如Student 类，
  - 实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

```
Michael = Student('Michael', 98) # 创建实例对象
Kristen = Student('Kristen', 93) # 创建实例对象

Michael.print_score() # 调用类的方法
Kristen.print_score() # 调用类的方法
```

```
Michael: 98
Kristen: 93
```

类和实例

- 仍以“处理学生成绩”举例，我们来说明类和实例的概念
  - Python中，定义类是通过class关键字
  - class 后为类名，类名通常是大写开头的单词
  - 类名（object），表示该类从object父类继承下来。通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

```
class Student2(object):
    pass
```

类和实例

- 仍以“处理学生成绩”举例，我们来说明类和实例的概念
  - 定义好了Student2 类，就可以根据它来创建出其实例

```
Michael = Student2()
Kristen = Student2()
```

```
print(Michael)
print(Kristen)
```

```
<__main__.Student2 object at 0x7fad4b7d5ca0>
<__main__.Student2 object at 0x7fad4b7d5730>
```

```
print(Student2)
```

```
<class '__main__.Student2'>
```

类和实例

- 仍以“处理学生成绩”举例，我们来说明类和实例的概念
  - 定义好了Student2 类，就可以根据它来创建出其实例

```
Michael = Student2()
Kristen = Student2()
```

```
print(Michael)
print(Kristen)
```

```
<__main__.Student2 object at 0x7fad4b7d5ca0>
<__main__.Student2 object at 0x7fad4b7d5730>
```

```
print(Student2)
```

```
<class '__main__.Student2'>
```

Student2 本身是一个类
而Michael指向的是一个Student2 的实例，0x7fef7387a160是其内存地址，每个object 的地址都不一样

类和实例

- 仍以“处理学生成绩”举例，我们来说明类和实例的概念
  - Student2类实例化后，可以自由地给一个实例变量绑定属性

```
Michael.name="Michael Simon"
print(Michael.name)
```

```
Michael Simon
```

```
Kristen.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-24-41b55b5ff35e> in <module>
----> 1 Kristen.name

AttributeError: 'Student' object has no attribute 'name'
```

类和实例

- 类起到模板的作用，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。
- 通过定义一个特殊的__init__方法（构造函数），在创建实例的时候，就把name，score等属性绑上去
  - __init__前后分别有两根下划线
  - __init__()的第一个参数永远是self，表示创建的实例本身。因此，在__init__()内部，就可把各种属性绑定到self，因为self就指向创建的实例本身。

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score
```

类和实例

- 有了__init__()，在创建实例时，就不能传入空的参数了，必须传入与__init__()相匹配的参数，但self不需要传，Python 解释器自己会把实例变量传进去。

```
Michael = Student('Michael', 98) # 创建实例对象
print(Michael.name)
print(Michael.score)
```

```
Michael
98
```

类和实例

- 有了__init__()，在创建实例时，就不能传入空的参数了，必须传入与__init__()相匹配的参数，但self不需要传，Python 解释器自己会把实例变量传进去。

```
Michael = Student('Michael', 98) # 创建实例对象
print(Michael.name)
print(Michael.score)
```

```
Michael
98
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且调用时，不用传递该参数。除此之外，类的方法和普通函数没有显著区别，所以，仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

人工智能与
Python程序设计

提纲

1. 类和实例
2. 数据封装
3. 访问限制

数据封装

- OOP 的一个重要特点就是数据封装。
  - Student类中，每个实例拥有各自的name和score数据
  - 可以通过函数来访问实例的数据，如打印某个学生的成绩

```
def print_score(std):
    print('%s: %d' % (std.name, std.score))

print_score(Michael)
```

```
Michael: 98
```

数据封装

- Student 的实例本身就拥有这些数据，故要访问它们，可以直接在 Student 类的内部定义访问数据的函数，进而把“数据”封装。
- 封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法。

数据封装

- Student 的实例本身就拥有这些数据，故要访问它们，可以直接在 Student 类的内部定义访问数据的函数，进而把“数据”封装。
- 封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法。

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))
```

数据封装

- Student 的实例本身就拥有这些数据，故要访问它们，可以直接在 Student 类的内部定义访问数据的函数，进而把“数据”封装。
- 封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法。

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))
```

我们发现，要定义一个方法，除了第一个参数是self外，其他和普通函数一样。

数据封装

- 要调用一个方法，只需要在实例变量上直接调用即可。除了self不用传递，其他参数正常传入。
  - 从调用方来看Student 类，只需在创建实例时给定name和score
  - score打印会在Student 类的内部实现。这些数据和逻辑被『封装』起来，调用会变得容易。

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
```

```
Michael Simon: 98
```

数据封装

- 通过封装，可以给class增加新的方法。

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数， self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

人工智能与
Python程序设计

提纲

1. 类和实例
2. 数据封装
3. 访问限制

访问限制

- 在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，从而隐藏了内部的复杂逻辑。
  - 从示例Student类的定义来看，外部代码可以修改一个实例的属性

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
Michael.score=80
Michael.print_score()
```

```
Michael Simon: 98
Michael Simon: 80
```

访问限制

- 在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，从而隐藏了内部的复杂逻辑。
  - 从示例Student类的定义来看，外部代码可以修改一个实例的属性
  - 为了让让内部属性不被外部访问，可以把属性的名称前加上两个下划线__，使其变为一个私有变量(private)，未加下划线则为公有变量(public)
  - 私有变量只有内部可以访问，外部不能访问；
  - 公有变量内部、外部皆可以访问

访问限制

```
class Student(object):

    def __init__(self, name, score):
        self.__name  = name   # 私有变量
        self.__score = score  # 私有变量

    def print_score(self):
        print('%s: %d' % (self.__name, self.__score))
```

```
Michael = Student('Michael', 98) # 创建实例对象
Michael.print_score()
print(Michael.__score)
```

```
Michael: 98
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-bb01d8fad8b6> in <module>
      1 Michael = Student('Michael', 98) # 创建实例对象
      2 Michael.print_score()
----> 3 print(Michael.__score)

AttributeError: 'Student' object has no attribute '__score'
```

访问限制

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

```
Michael = Student('Michael', 98) # 创建实例对象
Michael.print_score()
Michael.__score = 80
Michael.print_score()
```

```
Michael: 98
Michael: 98
```

访问限制

- 通过引入私有变量，可以确保外部代码不能随意修改对象内部的状态。即，通过访问限制的保护，使得代码更加健壮。
- 如果外部代码想访问、修改私有变量，可通过增加类的方法进行实现。

```
class Student(object):
    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def set_score(self, score):
        self.__score = score
```

访问限制

那么，类的方法会比直接通过外部访问/修改有什么优势呢？

```
class Student(object):
    ...

    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

在类的方法中，可以对参数做相关的检查，避免出现异常错误。

访问限制

- 注意：
  - 类似于__xxx的变量是private变量，外部代码不能直接访问。
  - 类似于_xxx的变量允许外部代码访问，但按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但请把我视为private 变量，不要随意访问”
  - 类似于__xxx__的变量是特殊变量，不是private 变量，外部代码可直接访问。

访问限制

- 再回来分析这个示例：

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
Michael.__score=80
Michael.print_score()
```

```
Michael Simon: 98
Michael Simon: 98
```

```
print(Michael._Student__score)
print(Michael.__score)
```

```
98
80
```

内部的__score变量已经被Python解释器自动改成了_Student__score，外部代码给Michael实例新增了一个__score变量

人工智能与
Python程序设计

回顾

1. 类和实例
2. 数据封装
3. 访问限制

谢谢！