

STAT243 Lecture 2 Data Technology

1 Data storage and file formats on a computer

Logic ▾

参考文献:

- Adler
- Nolan and Temple Lang, XML and Web Technologies for Data Sciences with R.
- Murrell, Introduction to Data Technologies.
- SCF tutorial: [Working with large datasets in SQL, R, and Python](#)

参考视频: bCourses Media Gallery 有 4 段 2020 年的参考视频 (使用 R):

- Text files and ASCII
- Encodings and UTF-8
- HTML
- XML and JSON

1.1 基本概念与术语

- **文件 (file)**: 按字节顺序组织的数据集合, 包含内容与少量元信息 (metadata)。
- **目录 (directory)**: 文件的层级组织方式, 路径使用分隔符表示层级。
- **路径 (path)**:
 - 绝对路径: 从根目录开始, 例如 `/home/user/data.csv`。
 - 相对路径: 相对当前工作目录, 例如 `../data/airline.csv`。
- **扩展名 (extension)**: 约定俗成的类型标识, 如 `.csv`, `.json`, `.parquet`, 不决定编码方式但有提示作用。
- **MIME type**: 互联网场景中文件类型的标准描述, 如 `text/csv`, `application/json`。
- **bit / byte**: bit 为二进制位, 取值 0/1; byte 为 8 bits。很多文件属性与编码的度量单位以 byte 计。

1.2 Text files vs Binary files

- **Text files**
 - 定义: 文件中的比特编码为单个字符; 数字以字符形式写入 (例如由数字字符 0–9 组成)。
 - 例子: CSV, XML, HTML, JSON。
 - 编码:
 - ASCII: 1 byte/char; 最多 128 个字符 (字母大小写、数字、标点与控制字符)。
 - UTF-8: 1–4 bytes/char。
 - 优点: 人可读、跨语言、易 diff。
 - 局限:
 - 某些格式 (如 JSON、HTML) 不易按“行”解释/操作, 不适合用 shell 行处理工具;
 - 存储冗余、解析耗时; 随机访问差。
- **Binary files**
 - 定义: 比特用自定义二进制布局编码信息, 而非直接对应字符; 程序需按格式解释字节语义。
 - 例子: netCDF、Python pickle、R `.Rda`、HDF5、已编译代码等。
 - 性能: 通常更节省空间且可随机访问; 数值通常以 8 bytes/数存储 (详见 Unit 8)。
 - 局限: 不可读、跨语言互操作依赖库与规范。

1.3 Common file types

说明：此处既包含文本格式也包含二进制/容器格式。

1. Flat text (定宽或分隔符)

- 一行一条记录；列为不同变量。
- 常见分隔符：tab、逗号、空格、pipe |。
- 常见扩展名：.txt, .csv。
- 字段中若包含分隔符或换行，可使用引号包裹；`pandas.read_table()` 能处理带引号情况。
- 换行差异与转换：
 - Windows 行尾，UNIX 行尾。
 - 在 UNIX 上看到行尾 ^M 表示混入了 DOS 行尾。
 - 可用 `fromdos / dos2unix` 转换为 UNIX；`todos / unix2dos` 转换为 DOS/Windows。

2. 逐行文本但无列含义

- 例如纯文本或部分生物信息学数据，一行仅是一段信息。

•

3. 数据交换格式

- XML, JSON 为自描述（元数据在文件中）。
- Python 常用库：`lxml`, `json`（细节见后续章节）。

•

4. HTML

- 常用于 web 抓取；结合 `requests`, `BeautifulSoup` 解析。

•

5. netCDF / HDF5

- 科学计算常用；以变量保存多维数组，包含维度与元数据。
- Python 可用 `netCDF4`。

•

6. 其他软件数据格式

- Stata/SAS/SPSS; `pandas.read_stata` / `read_spss` / `read_sas` 可读入。

•

7. Excel

- 不建议作为数据分发格式：
 - 专有且结构复杂、格式可能变化；
 - 行数等限制；
 - 难以用 UNIX 工具处理；
 - 常含多工作表、图表、宏等非数据内容。
- 建议：在 Excel 导出为 CSV 再读入 Python。

•

8. Databases

- Python 可与 SQLite, DuckDB, PostgreSQL, MySQL, Oracle 等交互。
- 通过 SQL 查询并将结果返回至 Python（详见后续单元与 tutorial）。

1.4 CSV vs. specialized formats such as Parquet

- **CSV 优点：**简单、人可读、便于用工具处理。
- **CSV 劣势：**
 - 行存储类型混杂，压缩效果一般；
 - 明文存储分隔符与换行，体积偏大；
 - 随机访问不便：定位第 N 行需顺序扫描换行符（如找第 10 行需先找到第 9 个换行）。

- **Parquet** 优势:

- 列式存储(列块), 同列类型一致且重复多, 便于编码与压缩;
- 可只读取所需列, I/O 更小, 通常更快; 常以多个文件分片存储。

- **示例 (原文给出):**

- 读取 CSV ≈ 1.815 s; 读取 Parquet ≈ 0.443 s。
- 大小: CSV ≈ 51 MB; Parquet ≈ 8 MB。
- 代码与命令:



Python

```
1 import time, os, pandas as pd
2 ## Read from CSV
3 t0 = time.time()
4 data_from_csv = pd.read_csv(os.path.join('..', 'data', 'airline.csv'))
5 print(time.time() - t0) ## ~1.815 s
6 ## Write Parquet
7 data_from_csv.to_parquet(os.path.join('..', 'data', 'airline.parquet'))
8 ## Read from Parquet
9 t0 = time.time()
10 data_from_parquet = pd.read_parquet(os.path.join('..', 'data', 'airline.parquet'))
11 print(time.time() - t0) ## ~0.443 s
```



Shell

```
1 ls -l ../data/airline.csv
2 ls -l ../data/airline.parquet
```

2 Reading data from text files into Python

2.1 Core Python functions

- 使用 Pandas 读取文本数据:

- `read_table` 与 `read_csv` 读取分隔符文本;
关键参数:
 - `sep` 分隔符
 - `header` 是否有表头。
- 定宽文本使用 `read_fwf()` 读取为 DataFrame。
- 类型推断: Pandas 会自动推断, 但更稳妥做法是通过 `dtype` 明确指定各列类型。
- `sep` 支持正则: 可以用正则表达式按空白或复杂分隔模式拆分。

- **示例 1:** `RTADataSub.csv` 初读入与缺失处理。



Python

```
1 import os, pandas as pd
2
3 ## 初次读取: 不指定 dtype 时, 多数列为 object
4 dat = pd.read_table(os.path.join('..', 'data', 'RTADataSub.csv'),
5                     sep = ',', header = None)
6 print(dat.dtypes.head())
7 ## Output:
8 ## 0    object
9 ## 1    object
10 ## 2   object
11 ## 3   object
12 ## 4   object
13 ## dtype: object
14
```

```

15 print(dat.loc[0,1])
16 ## Output:
17 ## 2336
18
19 print(type(dat.loc[0,1]))
20 ## Output:
21 ## <class 'str'>
22
23 ## 将 'x' 视为缺失
24 dat2 = pd.read_table(os.path.join('..', 'data', 'RTADataSub.csv'),
25                      sep = ',', header = None, na_values = 'x')
26 print(dat2.dtypes.head())
27 ## Output:
28 ## 0    object
29 ## 1    float64
30 ## 2    float64
31 ## 3    float64
32 ## 4    float64
33 ## dtype: object
34
35 print(dat2.loc[:,1].unique())
36 ## Output:
37 ## array([2336.,    nan,  1450., ...])

```

- 示例 2: `hiveSequ.csv` 中通过 `dtype` 精确控制多列类型。

 Python

```

1 dat = pd.read_table(os.path.join('..', 'data', 'hiveSequ.csv'),
2                      sep = ',', header = 0,
3                      dtype = {
4                          'PatientID': int,
5                          'Resp': int,
6                          'PR Seq': str,
7                          'RT Seq': str,
8                          'VL-t0': float,
9                          'CD4-t0': int
10                         })
11 print(dat.dtypes)
12 ## Output:
13 ## PatientID      int64
14 ## Resp           int64
15 ## PR Seq        object
16 ## RT Seq        object
17 ## VL-t0         float64
18 ## CD4-t0        int64
19 ## dtype: object
20
21 print(dat.loc[0, 'PR Seq'])
22 ## Output:
23 ## CCTCAAATCACTTT...

```

- 只读取部分列: 使用 `usecols` 指定需要的列可以减少内存与 I/O。
- 读入前检查: 建议先在终端用 `less` 等预览原始文件, 以便发现分隔符、缺失标记、列数不齐等问题。
- 若文件列宽不齐 (ragged lines): 可以先逐行读为字符串, 再按切片处理固定位置字段。

 Python

```

1 file_path = os.path.join('..', 'data', 'precip.txt')
2 with open(file_path, 'r') as file:

```

```

3     lines = file.readlines()
4
5     id = [line[3:11] for line in lines]
6     year = [int(line[17:21]) for line in lines]
7     month = [int(line[21:23]) for line in lines]
8     nvalues = [int(line[27:30]) for line in lines]
9     print(year[0:5])
10    ## Output:
11    ## [2010, 2010, 2010, 2010, 2010]

```

- 上述 `precip.txt` 实际为 fixed-width, 使用 `pandas.read_fwf()` 更合适。

 **Remark** ▾

`with` 语句是标准的 Python 文件上下文管理方式, 离开代码块会自动关闭文件句柄。

2.2 Connections and streaming

- 除了普通文件, 还可以从不同连接读取: 压缩文件、归档、子进程输出、网络资源等。



Python

```

1 import gzip
2 with gzip.open('dat.csv.gz', 'r') as file:
3     lines = file.readlines()
4     ## Output:
5     ## [b'...first line...', b'...'] ## 示例
6
7 import zipfile
8 with zipfile.ZipFile('dat.zip', 'r') as archive:
9     with archive.open('data.txt', 'r') as file:
10        lines = file.readlines()
11     ## Output:
12     ## [b'...first line...', b'...'] ## 示例
13
14 import subprocess, io
15 command = "ls -al"
16 output = subprocess.check_output(command, shell = True)
17 with io.BytesIO(output) as stream:
18     content = stream.readlines()
19     ## Output:
20     ## [b'total ...', b'drwxr-xr-x ...', ...] ## 示例
21
22     ## 直接从 URL 读取 (制表符分隔)
23 df = pd.read_csv("https://download.bls.gov/pub/time.series/cu/cu.item", sep="\t")
24 print(df.shape)
25     ## Output:
26     ## (N, M) ## 示例: 行列尺寸

```

- 流式/分块读取 (online processing / streaming / chunking): 适用于大文件。



Python

```

1 file_path = os.path.join('..', 'data', 'RTADataSub.csv')
2 chunksize = 50
3 with pd.read_csv(file_path, chunksize = chunksize) as reader:
4     for chunk in reader:
5         print(f'Read {len(chunk)} rows.')
6         ## Output:

```

```
## Read 50 rows.
```

- 将字符串当作类文件对象读取，便于对接只接受 file-like 的 API。



Python

```

1 file_path = os.path.join('..', 'data', 'precip.txt')
2 with open(file_path, 'r') as file:
3     text = file.read()
4
5 stringIOtext = io.StringIO(text)
6
7 df = pd.read_fwf(stringIOtext, header = None, widths = [3,8,4,2,4,2])
8 print(df.head().shape)
9 ## Output:
10 ## (5, 6) ## 示例: 前五行的形状

```

- 也可以创建连接用于写出，但需要先显式打开连接（写出在下一节）。

2.3 File paths

- 避免在代码中硬编码绝对路径，改用项目内的相对路径。



Python

```

1 dat = pd.read_csv('../data/cpds.csv')
2 print(dat.shape)
3 ## Output:
4 ## (N, M) ## 示例

```

- 使用 UNIX 风格分隔符具有跨平台性；Windows 风格 \ 在 Linux/Mac 上不可用。



Python

```

1 ## good: 跨平台
2 pd.read_csv('../data/cpds.csv')
3 ## Output:
4 ## (N, M) ## 示例
5
6 ## bad: 在 Linux/Mac 上不可用
7 pd.read_csv('..\data\cpds.csv')
8 ## Output:
9 ## FileNotFoundError: [Errno 2] No such file or directory: '..\\data\\cpds.csv' ## 示例

```

- 更好的方法是：使用 `os.path.join` 构造不依赖于操作系统的路径。



Python

```

1 pd.read_csv(os.path.join('..', 'data', 'cpds.csv'))

```

2.4 Reading data quickly: Arrow and Polars

- Apache Arrow：列式内存布局，Python 通过 PyArrow 调用。
 - 同列值连续存放，支持高效访问。
 - 可从 Parquet、Arrow 原生格式、文本读取。
 - 按需读取，避免整表常驻内存。
- 其他避免全量入内存方案：`Dask`、`numpy.load(mmap_mode=...)`。
- `polars`：强调速度的 DataFrame 库，常见对比示例如下。



Python

```
1 import polars, time
2
3 t0 = time.time()
4 dat = pd.read_csv(os.path.join('..', 'data', 'airline.csv'))
5 t1 = time.time()
6
7 dat2 = polars.read_csv(os.path.join('..', 'data', 'airline.csv'), null_values = ['NA'])
8 t2 = time.time()
9 print(f"Timing for Pandas: {t1-t0}.")
10 ## Output:
11 ## Timing for Pandas: 0.99.
12 print(f"Timing for Polars: {t2-t1}.")
13 ## Output:
14 ## Timing for Polars: 0.91.
```

3 Output from Python

3.1 Writing output to files

- 与读取函数一一对应的**文本输出**接口：

- `open(file_path, 'w')` 写入新文件，`'a'` 追加写入。
- `file.write(str_obj)` 写入单个字符串；返回写入的字符数。
- `file.writelines(lines)` 批量写入字符串序列（不会自动加换行）。
- 建议配合 `with` 使用，确保句柄自动关闭。



Python

```
1 import os
2 file_path = os.path.join('/tmp', 'tmp.txt')
3 with open(file_path, 'w') as file:
4     n = file.write('hello\n')
5     print(n)
6 ## 输出:
7 ## 6
8
9 lines = ['alpha\n', 'beta\n']
10 with open(file_path, 'a') as file:
11     file.writelines(lines)
```

- **Pandas** 输出：

- `DataFrame.to_csv(path, index=False)` 输出为 CSV。
- `DataFrame.to_parquet(path)` 输出为 Parquet（列式，体积更小，读取更快）。



Python

```
1 import pandas as pd
2
3 df = pd.DataFrame({'x': [1, 2], 'y': ['a', 'b']})
4 print(df.to_csv(index=False))
5 ## 输出:
6 ## x,y\n
7 ## 1,a\n
8 ## 2,b\n
```

- **JSON** 输出：

- `json.dump(obj, fp)` 将 Python dict/list 等序列化为 JSON 内容写入文件。

- 适用于在程序间交换结构化对象（可与 Web API 互操作）。



Python

```

1 import json, os
2 obj = {'a': 1, 'b': [2, 3]}
3 path = os.path.join('/tmp', 'obj.json')
4 with open(path, 'w') as f:
5     json.dump(obj, f)
6 print(os.path.exists(path))
7 ## 输出:
8 ## True

```

- Pickle (二进制序列化):**

- `pickle.dump(obj, fp)` / `pickle.load(fp)`；跨平台、可传输复杂 Python 对象。
- 注意安全：只从[可信来源](#)加载 pickle。



Python

```

1 import pickle, os
2 s = {'k': (1, 2, 3)}
3 path = os.path.join('/tmp', 'obj.pkl')
4 with open(path, 'wb') as f:
5     pickle.dump(s, f)
6 print(os.path.getsize(path) > 0)
7 ## 输出:
8 ## True

```

3.2 Formatting output

- 使用 **format 迷你语言** 控制对齐、宽度与小数位：

- `{:>10}` 右对齐占 10 列；`{:.10f}` 固定 10 位小数。
- 函数式接口 `format(value, '15.10f')` 与等价的 `'{:15.10f}'.format(value)`。



Python

```

1 print('{:>10}'.format(3.5))
2 ## 输出:
3 ##      3.5
4
5 print('{:.10f}'.format(1/3))
6 ## 输出:
7 ## 0.3333333333
8
9 print('{:15.10f}'.format(1/3))
10 ## 输出:
11 ##      0.3333333333
12
13 print(format(1/3, '15.10f'))
14 ## 输出:
15 ## 0.3333333333

```

- f-strings (Python 3.6+):** 在字符串字面量中直接插入表达式。



Python

```

1 val1, val2 = 1.5, 2.5
2 print(f"Let's add {val1} and {val2}.")
3 ## 输出:

```

```
4 ## Let's add 1.5 and 2.5.
```

- 旧式 `%` 格式化：保留以兼容历史代码；`%s` 字符串、`.5f` 小数位、`15.7f` 宽度与精度。

```
Python  
1 num1 = 1/3  
2 print("Let's add the %s numbers %.5f and %15.7f." % ('floating point', num1, 32+1/7))  
3 ## 输出：  
4 ## Let's add the floating point numbers 0.33333 and 32.1428571.
```

- 将格式化结果写入文件：

```
Python  
1 import os  
2 file_path = os.path.join('/tmp', 'tmp.txt')  
3 with open(file_path, 'a') as file:  
4     m = file.write("Let's add the %s numbers %.5f and %15.7f.\n" % ('floating point', num1,  
5 ,32+1/7))  
6     print(m)  
7 ## 输出：  
8 ## 55
```

- `round()` 也可控制显示位数，但更推荐直接用格式化确保输出可控。

4 Working with Information on the Web

Logic

- 以编程方式与 Web 交互：以下载为主，也可上传
- 常见格式：HTML, XML, JSON, YAML。
- 核心工具：`requests`、`BeautifulSoup` / `lxml`、`json`、`yaml`、`pandas.read_html` / `read_json`。

4.1 Reading HTML (抓取 HTML 表格)

- 关键思路：获取 HTML 源码 → 解析 DOM → 抽取 `<table>` 或节点文本。
- 浏览器中可先查看 **View Source / Developer Tools** 了解结构与是否存在大量 Javascript。

4.1.1 使用 `find_all` 通过 HTML 标签或属性搜索

```
Python  
1 import io, requests, pandas as pd  
2 from bs4 import BeautifulSoup as bs  
3  
4 URL = "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population"  
5 user_agent = "stat243_educational_bot/0.1 (paciorek@berkeley.edu)"  
6 headers = {'User-Agent': user_agent}  
7 response = requests.get(URL, headers=headers)  
8 html = response.content  
9  
10 soup = bs(html, 'html.parser')  
11 html_tables = soup.find_all('table')  
12  
13 ## pandas.read_html 需要字符串/文件句柄，不直接接受 Tag 对象  
14 pd_tables = [pd.read_html(io.StringIO(str(tbl)))[0] for tbl in html_tables]  
15 [x.shape for x in pd_tables]
```

```

16 ## 输出:
17 ## [(242, 6), (13, 2), (1, 2)]
18
19 pd_tables[0].head()
20 ## 输出:
21 ##      Location ... Notes
22 ## 0      World ...   NaN
23 ## 1      India ... [b]
24 ## ...

```

- `BeautifulSoup` 构建树后可按标签或属性搜索，再交给 `pandas.read_html` 解析。

4.1.2 Extracting hyperlinks (提取超链接)



Python

```

1 import requests
2 from bs4 import BeautifulSoup as bs
3
4 URL = "http://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year"
5 response = requests.get(URL)
6 soup = bs(response.content, 'html.parser')
7
8 ## 方法 1: 所有 <a> 标签
9 a_elements = soup.find_all('a')
10 links1 = [x.get('href') for x in a_elements]
11
12 ## 方法 2: 具有 href 属性的 <a>
13 href_elements = soup.find_all('a', href=True)
14 links2 = [x.get('href') for x in href_elements]
15
16 links2[:9]
17 ## 输出:
18 ## ['?C=N;O=D', '?C=M;O=A', '?C=S;O=A', '?C=D;O=A', '/pub/data/ghcn/daily/',
19 ## '1750.csv.gz', '1763.csv.gz', '1764.csv.gz', '1765.csv.gz']

```

4.1.3 CSS selectors (用选择器抽取)



Python

```

1 ## 所有 <tr> 内部的 <th>
2 soup.select("tr th")
3 ## 输出:
4 ## [<th>...</th>, <th>...</th>, ...]
5
6 ## 所有父元素为 <th> 的 <a>
7 soup.select("th > a")
8 ## 输出:
9 ## [<a href="?C=N;O=D">Name</a>, <a href="?C=M;O=A">Last modified</a>, ...]

```

4.1.4 XPath (用 lxml 进行 XPath 查询)



Python

```

1 import lxml.html
2
3 ## 将 BeautifulSoup object 转为 lxml object
4 lxml_doc = lxml.html.fromstring(str(soup))
5
6 ## 所有带 href 的 <a>

```

```

7 a_elements = lxml_doc.xpath('//a[@href]')
8 links = [x.get('href') for x in a_elements]
9 links[:9]
10 ## 输出:
11 ## ['?C=N;O=D', '?C=M;O=A', '?C=S;O=A', '?C=D;O=A', '/pub/data/ghcn/daily/',
12 ## '1750.csv.gz', '1763.csv.gz', '1764.csv.gz', '1765.csv.gz']

```

4.2 XML, JSON, and YAML

- 三者均支持键值、数组与层级结构
- 读取需用对应库解析为 Python 结构 (dict/list 等)。

4.2.1 XML (结构化自描述)

- XML 是一种以自描述格式储存数据的 markup language, 通常有 hierarchical structure, 不需要 metadata
- XML 文档具有树状结构, 由元素 (节点) 组成
- 常见存档/办公文档/空间信息 (如 KML)。

 XML

```

1 <?xml version="1.0"?>
2 <catalog>
3   <book id="bk101">
4     <author>GambardeLLA, Matthew</author>
5     <title>XML Developer's Guide</title>
6     <genre>Computer</genre>
7     <price>44.95</price>
8     <publish_date>2000-10-01</publish_date>
9     <description>An in-depth look at creating applications with XML.</description>
10    </book>
11    <book id="bk102">
12      <author>Ralls, Kim</author>
13      <title>Midnight Rain</title>
14      <genre>Fantasy</genre>
15      <price>5.95</price>
16      <publish_date>2000-12-16</publish_date>
17      <description>A former architect battles corporate zombies, an evil sorceress, and her
own childhood to become queen of the world.</description>
18    </book>
19  </catalog>

```

Example ▾

示例: Kiva 最新贷款数据 (注意: 在线接口可能返回 403, 需本地保存演示)。我们采用两种方法:

1. 暴力解法 (将数据视作列表而不是树结构)
2. 使用 XPath 来遍历树结构

 Python

```

1 import xmltodict
2 ## 假设 'newest.xml' 已手动下载 (或用浏览器另存)
3 with open('newest.xml', 'r') as file:
4     content = file.read()
5
6 ## 一些 XML 含有裸 '&', 需先替换以避免解析错误
7 content = content.replace("&", "and")
8 data = xmltodict.parse(content)

```

```

9
10 data.keys()
11 ## 输出:
12 ## dict_keys(['response'])
13
14 data['response'].keys()
15 ## 输出:
16 ## dict_keys(['paging', 'loans'])
17
18 len(data['response']['loans']['loan'])
19 ## 输出:
20 ## 20
21
22 type(data['response']['loans']['loan'][2])
23 ## 输出:
24 ## dictionary
25
26 data['response']['loans']['loan'][2]['activity']
27 ## 输出:
28 ## 'Retail'

```



Python

```

1 ## 同样可用 lxml + XPath 提取指定字段
2 from lxml import etree
3
4 doc = etree.fromstring(content)
5 loans = doc.xpath("//loan")
6 [loan.xpath("activity/text()") for loan in loans][:3]
7 ## 输出:
8 ## [['Poultry'], ['Retail'], ['Retail']]
9
10 #### 假设只想要 country locations of the loans (using XPath)
11 [loan.xpath("location/country/text()") for loan in loans]
12 ## 输出:
13 ## [['Uganda'], ['Ecuador'], ['Ecuador'] ...]
14
15 #### or extract the geographic coordinates
16 [loan.xpath("location/geo/pairs/text()") for loan in loans]
17 ## 输出:
18 ## [[ '-0.352537 31.552699'], ['-1.054723 -80.45249'] ...]

```

4.2.2 JSON (层级键值, 较 XML 简洁)

- JSON 文件以 “**attribute-value** pairs” (也称为“键-值”对) 结构化, 通常具有层次结构
- 可以使用 `json` 包将 JSON 读取到 Python 中
- JSON 的最外层可以是 **对象 (object, 用花括号 {})** 或 **数组 (array, 用方括号 [])**。如果最外层是数组, 那么就没有键 (key) 去命名它的元素。



JSON

```

1 {
2     "firstName": "John",
3     "lastName": "Smith",
4     "isAlive": true,
5     "age": 25,
6     "address": {
7         "streetAddress": "21 2nd Street",
8         "city": "New York",
9         "state": "NY",

```

```
10     "postalCode": "10021-3100"
11 },
12 "phoneNumbers": [
13     { "type": "home", "number": "212 555-1234" },
14     { "type": "office", "number": "646 555-4567" }
15 ],
16 "children": [],
17 "spouse": null
18 }
```



Python

```
1 import json
2 ## 假设 'newest.json' 已手动下载
3 with open('newest.json', 'r') as file:
4     content = file.read()
5
6 data = json.loads(content)
7 list(data.keys())
8 ## 输出:
9 ## ['loans']
10
11 type(data['loans']), data['loans'][0]['location']['country']
12 ## 输出:
13 ## (<class 'list'>, 'Uganda')
14
15 [c['location']['country'] for c in data['loans'][:5]
16 ## 输出:
17 ## ['Uganda', 'Ecuador', 'Ecuador', 'Tajikistan', 'Mali']
```

⚠ Remark ▾

注意：JSON 不原生支持缺失、无穷大等特殊值。

4.2.3 YAML (常用于配置)

- 以缩进表达层级，人类可读；缩进易出错；部分关键字（如 `on`）在某些实现中会被当作布尔。



YAML

```
1 name: deploy-book
2
3 ## Only run this when the master branch changes
4 on:
5   push:
6     branches:
7       - main
8
9 ## This job installs dependencies, build the book, and pushes it to `gh-pages`
10 jobs:
11   deploy-book:
12     runs-on: ubuntu-latest
13     steps:
14       - uses: actions/checkout@v2
15
16     ## Install dependencies
17     - name: Set up Python 3.9
18       uses: actions/setup-python@v1
19       with:
```

```
20     python-version: 3.9
21
22     - name: Install dependencies
23       run: |
24         pip install -r book-requirements.txt
```



Python

```
1 import yaml
2 with open("book.yml") as stream:
3     config = yaml.safe_load(stream)
4     print(config)
5     ## 输出:
6     ## {'name': 'deploy-book', 'True': {'push': {'branches': ['main']}}, 'jobs': {'deploy-book': ...}}
7
8     print(config.get('name'))
9     ## 输出:
10    ## deploy-book
11
12    len(config['jobs']['deploy-book']['steps'])
13    ## 输出:
14    ## 3
```

⚠ Remark ▾

注意 `on` 会被视作 boolean value

4.3 Web APIs and webscraping

- 目标：获取 Web 数据时，优先使用正式 API；当 API 不可用时，再考虑 webscraping，并遵守网站条款与伦理。
- 常用库：`requests`, `json`, `pandas`, `BeautifulSoup`, `lxml`, `yaml`。
- 核心操作路径：理解 HTTP → REST 风格 API → 参数拼接与分页 → 处理压缩与归档 → POST 与认证 → 第三方封装 → 动态页面。

4.3.1 What is HTTP?

- **请求方法**：常用 GET、POST、PUT、DELETE，实际数据提取以 GET 为主。
- **状态码**：200 成功，4xx 客户端错误（如 403/404），5xx 服务器错误。
- **URL 查询字符串**：? 之后为参数，& 分隔键值对，空格常编码为 + 或 %20。

☰ Example ▾

1. `www.somewebsite.com?param1=arg1¶m2=arg2`
2. `https://www.yelp.com/search?find_desc=plumbers&find_loc=Berkeley+CA&ns=1`

- **响应内容**：通常包含文本形式的内容（例如，HTML、XML、JSON）或原始字节



Python

```
1 import requests
2
3 url = "https://httpbin.org/status/200"
4 r = requests.get(url)
5 print(r.status_code)
6     ## 输出:
7     ## 200
```

4.3.2 APIs: REST-based web services

Logic

- 理想情况下，一个网络服务会用提供其 API (Application Programming Interface) 文档，该接口用于提供数据或允许其他交互
- REST 是一种流行的 API 标准/风格

- 资源导向**: 以 URL (也叫 endpoint) 作为资源，通过 query string 过滤，通过 `GET` 实现 request，常返回 JSON。
- 分页与每页条数**: 注意 `page`, `per_page`, `limit`, `offset` 等字段。
- 两种构造 request 的方式**:
 - 直接拼接查询字符串。
 - 使用 `params` 以 dict 传参。



Python

```
1 import json, requests
2
3 ## 方法一: 直接拼接查询字符串 (World Bank 示例)
4 url = "https://api.worldbank.org/V2/country?incomeLevel=MIC&format=json"
5 resp = requests.get(url)
6 data = json.loads(resp.content)
7
8 ### 注意 data truncation/pagination
9 if False:
10     url = "https://api.worldbank.org/V2/country?incomeLevel=MIC&format=json&per_page=1000"
11     response = requests.get(url)
12     data = json.loads(response.content)
13
14 ## 方法二: Programmatic control
15 baseURL = "https://api.worldbank.org/V2/country"
16 group = 'MIC'
17 format = 'json'
18 args = {'incomeLevel': group, 'format': format, 'per_page': 1000}
19 url = baseURL + '?' +
20     '&'.join(['='.join([key, str(args[key])])
21             for key in args])
22 response = requests.get(url)
23 data = json.loads(response.content)
24
25 ## 方法三: params 传参 (更稳妥)
26 baseURL = "https://api.worldbank.org/V2/country"
27 params = {"incomeLevel": "MIC", "format": "json", "per_page": 1000}
28 resp = requests.get(baseURL, params=params)
29 data = resp.json()
30
31 print(type(data), len(data))
32 ## 输出:
33 ## <class 'list'> 2
34
35 print(len(data[1]), isinstance(data[1][5], dict), data[1][5]['name'])
36 ## 输出:
37 ## 104 True Benin
```



Python

```
1 ## 简单的分页遍历 (若 API 需要翻页)
2 all_rows = []
```

```

3 page = 1
4 while True:
5     params = {"incomeLevel": "MIC", "format": "json", "per_page": 100, "page": page}
6     resp = requests.get(baseURL, params=params)
7     data = resp.json()
8     rows = data[1]
9     if not rows:
10        break
11     all_rows.extend(rows)
12     page += 1
13
14 print(len(all_rows) >= 100)
15 ## 输出:
16 ## True

```

4.3.3 HTTP requests by deconstructing an (undocumented) API

- 思路：用浏览器 DevTools 的 Network 面板观察实际请求（URL、headers、查询参数、cookies），在代码中复现。
- 典型场景：下载链接返回 zip 压缩，需内存解压后再读取 CSV。

 Python

```

1 import io, zipfile, requests, pandas as pd
2
3 itemCode = 526
4 baseURL = "https://data.un.org/Handlers/DownloadHandler.ashx"
5 yrs = ",".join(str(yr) for yr in range(2012, 2018))
6 filter_ = f"?DataFilter=itemCode:{itemCode};year:{yrs}"
7 args1 = "&DataMartId=FAO&Format=csv&c=2,3,4,5,6,7&"
8 args2 = "s=countryName:asc;elementCode:asc;year:desc"
9 url = baseURL + filter_ + args1 + args2
10
11 resp = requests.get(url)
12 ## 把 zip 文件放在内存中，而不是保存成 .zip 文件
13 with io.BytesIO(resp.content) as stream:
14     ## 在内存中打开这个 zip 文件
15     with zipfile.ZipFile(stream, "r") as archive:
16         ## 从压缩包中读取第一个文件，并用 pandas 加载
17         name = archive.filelist[0].filename
18         with archive.open(name, "r") as f:
19             dat = pd.read_csv(f)
20
21 print(dat.head(2))
22 ## 输出:
23 ##   Country or Area Element Code ... Value Value Footnotes
24 ## 0      Afghanistan      432    ...  202.19          NaN
25 ## 1      Afghanistan      432    ...   27.45          NaN

```

4.3.4 Webscraping ethics and best practices

- 是否应该抓：优先使用公开下载文件或正式 API；抓取是下策。
- 是否允许抓：遵守网站条款与 `robots.txt`；尊重速率限制与版权、隐私。
- 实践建议：请求加 `User-Agent`，缓存响应避免重复请求；对高频请求使用 `time.sleep`；谨慎处理认证信息。

 Python

```

1 import time, requests
2
3 headers = {"User-Agent": "stat243_educational_bot/0.1 (contact@example.com)"}
4 for page in range(1, 4):

```

```
5     r = requests.get("https://httpbin.org/get", params={"page": page}, headers=headers,
6     timeout=10)
7     print(r.status_code, r.json()["args"]["page"])
8     ## 输出:
9     ## 200 1
10    ## 200 2
11    ## 200 3
12    time.sleep(1) ## 友好限速
```

4.3.5 More details on HTTP requests

- **结构化参数**: `params` 适合 GET, 避免手写拼接错误。
- **复杂下载**: 大文件/二进制内容可用 `stream=True` 分块下载。
- **错误处理**: `response.raise_for_status()`, 或根据 `status_code` 分支处理。

 Python

```
1 ## 分块下载 (示意)
2 import requests
3
4 url = "https://speed.hetzner.de/100MB.bin"
5 with requests.get(url, stream=True) as r:
6     r.raise_for_status()
7     total = 0
8     for chunk in r.iter_content(chunk_size=8192):
9         if chunk:
10             total += len(chunk)
11     print(total > 0)
12 ## 输出:
13 ## True
```

4.3.6 POST example (创建 GitHub issue)

- **说明**: 需个人 access token; 最小权限原则; 切勿把 token 提交到仓库。
- **两种方式**: 裸 `requests.post` 与封装库 `PyGitHub`。

 Python

```
1 import requests
2
3 with open(".github-access-token.txt", "r") as file:
4     ghtoken = file.read().strip()
5
6 owner, repo = "paciorek", "test"
7 url = f"https://api.github.com/repos/{owner}/{repo}/issues"
8 issue = {
9     "title": "This is an example issue",
10    "body": "This is the body of the issue created via API."
11 }
12 headers = {
13     "Authorization": f"token {ghtoken}",
14     "Accept": "application/vnd.github+json"
15 }
16 resp = requests.post(url, json=issue, headers=headers)
17 print(resp.status_code in (200, 201))
18 ## 输出:
19 ## True
```

 Python

```

1 ## 使用 PyGitHub 封装
2 from github import Github
3
4 with open(".github-access-token.txt", "r") as file:
5     ghtoken = file.read().strip()
6
7 g = Github(ghtoken)
8 repo = g.get_repo("paciorek/test")
9 issue = repo.create_issue(
10     title="Test Issue Created Programmatically",
11     body="This is an issue filed programmatically using PyGitHub."
12 )
13 print(f"## {issue.number}", issue.html_url)
14 ## 输出:
15 ## ##18 https://github.com/paciorek/test/issues/18
16 g.close()

```

4.3.7 Accessing dynamic pages

- 适用场景：内容需 Javascript 渲染或需要模拟用户交互。
- 方案：`selenium` 驱动浏览器；或 `scrapy` 框架配合 `splash` 渲染。
- 注意：渲染成本高、速率慢，更应重视限速、重试与缓存；若站点有公开接口，优先 API。

5 File and String Encodings

5.1 File and string encodings

Logic

文本数据的本质是字符 \leftrightarrow 数值码位 \leftrightarrow 字节序列 之间的映射。理解 ASCII、Unicode、UTF-8 的关系，能系统性解决“乱码”“解码失败”等常见问题。

5.1.1 ASCII 基础与十六进制表示

- ASCII 含 $2^7 = 128$ 个字符与控制码，基本等同于 US 键盘字符集。每字符占 1 byte (8 位)。
- 为了方便表示，常以 2 个十六进制数表示 1 个字节。例如 '`M`' 的二进制 `01001101` 即十六进制 `0x4d`。（`0x` 表示 16 进制）
- 在 Python 中可以手工写入 ASCII 字节到文件，再读回验证。

 Python

```

1 ## 用十六进制字节写入 ASCII 文本 "Mom\n"
2 hexvals = b'\x4d\x6f\x6d\x0a'
3     ## \x 表示后面是一个16进制编码的字节
4     ## b 表示这是一个字节串 bytes object
5
6 with open('tmp.txt', 'wb') as textfile:
7     nbytes = textfile.write(hexvals)      ## 会返回写入的字节数
8 print(nbytes)
9 ## 输出:
10 ## 4

```

 Python

```

1 subprocess.run(["ls", "-l", "tmp.txt"], capture_output=True).stdout
2 ## 输出:
3 ## b'-rw-r--r-- 1 paciorek scfstaf 4 Sep  3 08:56 tmp.txt\n'
4

```

```
5 with open('tmp.txt', 'r') as textfile:  
6     line = textfile.readlines()  
7  
8     ## 输出:  
9     ## ['Mom\n']
```

5.1.2 Unicode 与 UTF-8：码位与编码

- **Unicode** 为字符分配**唯一整数码位** (code point)。Python 中 `str` 持有 Unicode 字符。
- **UTF-8** 是将 Unicode 码位编码为**变长字节序列**的通用方案：ASCII 仍为 1 byte，其他多为 2–4 bytes。

⚠ Remark ▾

- Unicode 是字符的“抽象编号系统”；
- UTF-8 是一种把这些编号“具体编码成字节”的方法。

Python

```
1 ## Python str 是 Unicode  
2 x2_unicode = 'Pe\u00f1a 3\u00f72' ## 包含 ñ 和 ÷  
3     ## \u 表示这是一个 Unicode 字符的码点  
4 print(x2_unicode)  
5 ## 输出：  
6 ## Peña 3÷2  
7  
8 print(type(x2_unicode))  
9 ## 输出：  
10 ## <class 'str'>
```

Python

```
1 ## 从字符到码位（整数）再到十六进制  
2 print(ord('ñ')) ## ñ 的 Unicode 编号是 U+00F1 = 241 (十进制)  
3 ## 输出：  
4 ## 241  
5  
6 print(hex(ord('ñ')))  
7 ## 输出：  
8 ## 0xf1  
9  
10 ## 查看 UTF-8 编码后的字节序列（十六进制转义）  
11 print(bytes('\u00f1', 'utf-8')) ## 'ñ' 在 UTF-8 中占用两个字节: `0xC3 0xB1`  
12 ## 输出：  
13 ## b'\xc3\xb1'  
14  
15 print(bytes('\u00f7', 'utf-8')) ## ÷  
16 ## 输出：  
17 ## b'\xc3\xb7'
```

Python

```
1 ## 直接写入 UTF-8 字节到文件，再从 shell 验证  
2 x2_utf8 = b'Pe\xc3\xb1a 3\xc3\xb72'  
3 with open('tmp2.txt', 'wb') as textfile:  
4     nbytes = textfile.write(x2_utf8)  
5 print(nbytes)  
6 ## 输出：  
7 ## 10
```



Shell

```
1 ## 文件大小与内容 (n-tilde 与 division symbol 各占 2 字节)
2 ls -l tmp2.txt
3 ## 输出:
4 ## -rw-r--r-- 1 user group 10 Sep 3 08:55 tmp2.txt
5
6 cat tmp2.txt
7 ## 输出:
8 ## Peña 3÷2
```

5.1.3 UTF-8 的 bit-wise representation 设计要点

- 兼容 ASCII：保证旧系统可直接读取。
- 避免混淆：短编码模式的比特不会出现在更长模式的内部，
- 前缀自描述：通过前导比特模式判断字符所占字节数，首位为 0 的字节即 ASCII。
- 实务含义：顺序扫描时可无歧义地解析字符边界；随机访问时仍需自前定位边界。

5.1.4 识别与转换：工具与参数

- UNIX 工具：
 - file path 可粗略识别文件类型与编码。
 - iconv -f src -t dst 可进行编码转换。
- Python：
 - 读文本时可在 open(..., encoding='utf-8') 显式声明编码。
 - 已在内存的 str 可用 .encode('utf-8' | 'latin1' | 'ascii' ...) 得到 bytes，实现编码转换。



Python

```
1 ## 显式指定读取编码
2 with open('file_nonascii.txt', 'r', encoding='latin1') as f:
3     lines = f.readlines()
4     print(lines[0][:40])
5 ## 输出:
6 ## (示例) 前 40 个字符...
```

5.1.5 Python 默认编码与 locale

- 现代 Python 默认源代码与 I/O 多以 UTF-8 为默认。
- 可用 locale.getlocale() 查看环境区域设置与默认编码。



Python

```
1 import locale
2 print(locale.getlocale())
3 ## 输出:
4 ## ('en_US', 'UTF-8')
```

5.1.6 变量名中的 Unicode（可读性与可移植性）

- Python、R、Julia 等均可在变量名中使用 Unicode 字符，但渲染与工具链支持可能不同。
- 建议在教学或多语言协作中谨慎使用，仅在上下文明确时采用。



Python

```
1 peña = 7 ## 变量名包含 ñ
2 print(peña)
3 ## 输出:
```

```
4 ## 7
5
6 ## 某些 PDF/终端不一定正确显示希腊字母等符号，注意渲染差异
```

5.1.7 编码转换示例与常见错误

- 同一 Unicode 文本经不同编码得到不同 bytes 表示； latin1 对西欧字符常更短，但字符集更窄。
- 将包含非 ASCII 字符的 str 用 'ascii' 编码会报错。



Python

```
1 text = 'Pe\u00f1a 3\u00f3' ## 'Peña 3÷2'
2
3 print(text.encode('utf-8'))
4 ## 输出:
5 ## b'Pe\xc3\xb1a 3\xc3\xb3'
6
7 print(text.encode('latin1'))
8 ## 输出:
9 ## b'Pe\xf1a 3\xf3'
10
11 try:
12     text.encode('ascii')
13 except Exception as error:
14     print(error)
15 ## 输出:
16 ## 'ascii' codec can't encode character '\xf1' in position 2: ordinal not in range(128)
```

⚠ Remark ▾

上例说明：其中 2 个非 ASCII 字符在 UTF-8 需各 2 bytes，而在 Latin-1 各 1 byte。Latin-1 覆盖约 191 个附加字符，主要是西欧语言的带重音字母等，但远不及 Unicode 完整。

5.1.8 UnicodeDecodeError 的定位与修复

- 典型症状：以默认 UTF-8 读取实际为 Latin-1（或其他编码）的文件会报错，例如：



Python

```
1 ## 错误示例：未显式声明编码
2 with open('file_nonascii.txt', 'r') as textfile:
3     lines = textfile.readlines()
4 ## 输出:
5 ## UnicodeDecodeError: 'utf-8' codec can't decode byte 0xac in position 7922: invalid start
byte
```

- 解决：**显式指定真实编码**。确认后重读即可。



Python

```
1 with open('file_nonascii.txt', 'r', encoding='latin1') as textfile:
2     lines = textfile.readlines()
3 print(lines[16925])
4 ## 输出:
5 ## 'from 5##cia7lw8lz2nX,%@ [128.32.244.179] by ncpc-email with ESMTP\n'
```

⌚ Logic ▾

排查顺序：先用 `file` 粗判 → 尝试以 UTF-8 读取 → 若失败，结合数据来源与地区试 `latin1` 等 → 使用 `iconv` 或 `.encode/.decode` 做转换。始终在边界上验证：抽样查看有无异常字符、混合换行或不可见控制符。

6 Data Structures

Logic

选数据结构会影响：内存占用、访问速度、增删是否需要大量复制、后续计算是否方便

6.1 Standard data structures in Python and R

- 常见：`dataframe`, `list`, `array/vector/matrix/tensor`。
- Python 常用 `numpy array`, `pandas dataframe` (来自额外包)。
- `dict`: 按键取值; R 可用 named vector、named list, 或更高效的 `environment`。
- R 若不是矩形数据或标准数值对象，常用（嵌套）`list`。
- 分布式数据结构在 Unit 7 讨论 (跨机器的数据)。

6.1.1 选用建议 (按任务)

- 表格数据处理 (分组、连接、透视) → `DataFrame`。
- 高效数值计算、线性代数 → `numpy array / R matrix`。
- 异质、层级数据 → `list`。
- 按键随机访问、映射 → `dict / environment`。
(以上结合课程描述整理，无额外延展。)

6.2 Other kinds of data structures

- 本类结构的差异点在于如何在元素间导航 (访问路径与更新方式)。

6.2.1 Set

- 定义：不含重复元素的集合；常用于去重与成员测试。



Python

```
1 s = {'a', 'b', 'b'}
2 print(s)
3 ## 输出:
4 ## {'a', 'b'}
5 print('a' in s)
6 ## 输出:
7 ## True
```

6.2.2 Linked list

- 结构：每个节点包含值与指向“下一个”的指针；双向链表还指向“上一个”。
- 优点：插入只需改相关指针，无需复制其他元素。
- 缺点：定位任意位置需要从头遍历。

6.2.3 Trees 与 Graphs

- 共同点：节点与边。
- `Tree`：父子层级结构 (也可含父指针)。
- `Graph`：边可无方向，允许出现环。

6.2.4 Stack 与 Queue

- **Stack**: 后进先出，只能直接访问栈顶；嵌套函数调用的行为就是栈，函数调用使用的内存称为 stack。
- **Queue**: 先进先出，类似排队。

Python

```
1 ## stack
2 stack = []
3 stack.append('a'); stack.append('b')
4 print(stack.pop())
5 ## 输出:
6 ## b
7
8 ## queue (用 deque)
9 from collections import deque
10 q = deque(['a','b'])
11 q.append('c')
12 print(q.popleft())
13 ## 输出:
14 ## a
```

- 使用方式：可直接实现，或通过 Python/R 的扩展包；R 中并不常用这些结构（但 tree/graph 应用广）。

6.3 Related concepts

6.3.1 Types

- 含义：信息如何存储、可做哪些操作。**Primitive types** 与底层存储紧密相关，如 boolean, integer, numeric, character, pointer。

6.3.2 Pointers

- 指向内存地址的引用；常用于避免不必要的复制。

6.3.3 Hashes

- 用哈希函数将 key 映射为地址，实现按键快速查找，避免 O(n) 线性扫描。

Python

```
1 ## 线性扫描 vs dict 查找
2 items = {'id': i, 'val': i*i} for i in range(5)
3 print([x for x in items if x['id'] == 3][0]['val'])
4 ## 输出:
5 ## 9
6 d = {x['id']: x['val'] for x in items}
7 print(d[3])
8 ## 输出:
9 ## 9
```