STAT243 Lecture 5.2 Interacting with the operating system and external code and configuring Python

1 与操作系统交互

> **Logic**
>
> Python 提供了多种与 operating system 交互的方式, 包括调用 shell 命令, 操作文件系统等

1.1 文件系统操作

常用函数:

- `os.getcwd()`: 获取当前工作目录
- `os.chdir()`: 切换目录
- `import`: 导入模块
- `pickle.dump()` / `pickle.load()`: 序列化与反序列化

1.2 调用 shell 命令

使用 `subprocess.run()` 执行 UNIX 命令并捕获输出:

Python

```python
import subprocess, io

result = subprocess.run(['ls', '-al'], capture_output=True)
print(result.stdout.decode())
```

使用 `io.BytesIO` 处理输出流:

Python

```python
with io.BytesIO(result.stdout) as stream:
    content = stream.readlines()
    print(content[:2])
```

1.3 文件系统查询

检查文件或目录是否存在:

Python

```python
os.path.exists("unit2-dataTech.qmd")  # True
```

列出目录内容:

Python

```python
os.listdir("../data")
# ['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz',
'airline.cube']
```

1.4 跨平台路径处理

使用 `os.path.join` 处理路径差异:

Python

```python
os.listdir(os.path.join("...", "data"))
# ['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz',
'airline.cube']
```

建议编写与操作系统无关的代码, 使用 `os.path.join` 等函数确保代码在不同平台上都能正常工作

1.5 系统信息获取

获取操作系统信息:

Python

```python
import platform
platform.system()  # 'Linux'

os.uname()
# posix.uname_result(sysname='Linux', nodename='smeagol', release='6.8.0-64-generic',
version='#67%20')
```

获取 Python 版本信息:

Python

```python
platform.python_version()  # '3.13.2'

sys.version
# '3.13.2 | packaged by conda-forge | (main, Feb 17 2025, 14:40:20) [GCC 13.3.0]'
```

1.6 环境变量

获取环境变量:

Python

```python
os.environ['PATH']
# '/system/linux/miniforge-3.13/bin:/system/linux/miniforge-
3.13/condabin:/system/linux/miniforge-3.13/bin/'
```

1.7 Python 脚本作为 shell 脚本

将 Python 脚本转换为可执行的 shell 脚本:

1. 在文本文件中编写 Python 代码, 例如 `example.py`
2. 在文件第一行添加 `#!/usr/bin/python` 或更具可移植性的 `#!/usr/bin/env python`
3. 使用 `chmod` 使文件可执行: `chmod ugo+x example.py`
4. 从命令行运行脚本: `./example.py`

1.8 命令行参数解析

使用 `argparse` 包处理命令行参数:

Python

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-y', '--year', default=2002, help='year to download')
parser.add_argument('-m', '--month', default=None, help='month to download')
args = parser.parse_args()
```

```
6  year = int(args.year)
```

运行方式:

Bash

```
1  ./example.py 2004 January
```

1.9 执行控制

使用 `Ctrl-C` 中断执行, 这会优雅地退出, 返回到命令未启动的状态 注意当 Python 超过可用内存时可能会有较长的延迟

2 与外部代码交互

像 R, Python, Julia 这样的脚本语言允许调用 "外部代码", 通常指 C 或 C++ (也包括 Fortran, Java 等其他语言)

在 R 和 Python 等通常比编译代码慢得多的语言中, 调用外部代码特别重要, 而在像 Julia 这样的快速语言中则不那么重要 (Julia 使用即时编译)

2.1 Python 中的外部代码调用

在 Python 中, 可以直接调用 C 或 C++ 代码, 或使用 **Cython** 与 C 交互

使用 Cython 可以:

- 在提供变量类型定义的情况下, 让 Cython 自动将 Python 代码转换为 C
- 定义可以从 Python 代码调用的 C 函数

2.2 R 中的外部代码调用

在 R 中, 可以使用 `.Call` 直接调用 C 或 C++ 代码, 或使用 **Rcpp** 包 Rcpp 专门设计用于编写感觉有点像 R 代码的 C++ 代码, 并且可以非常容易地在 R 和 C++ 之间传递数据