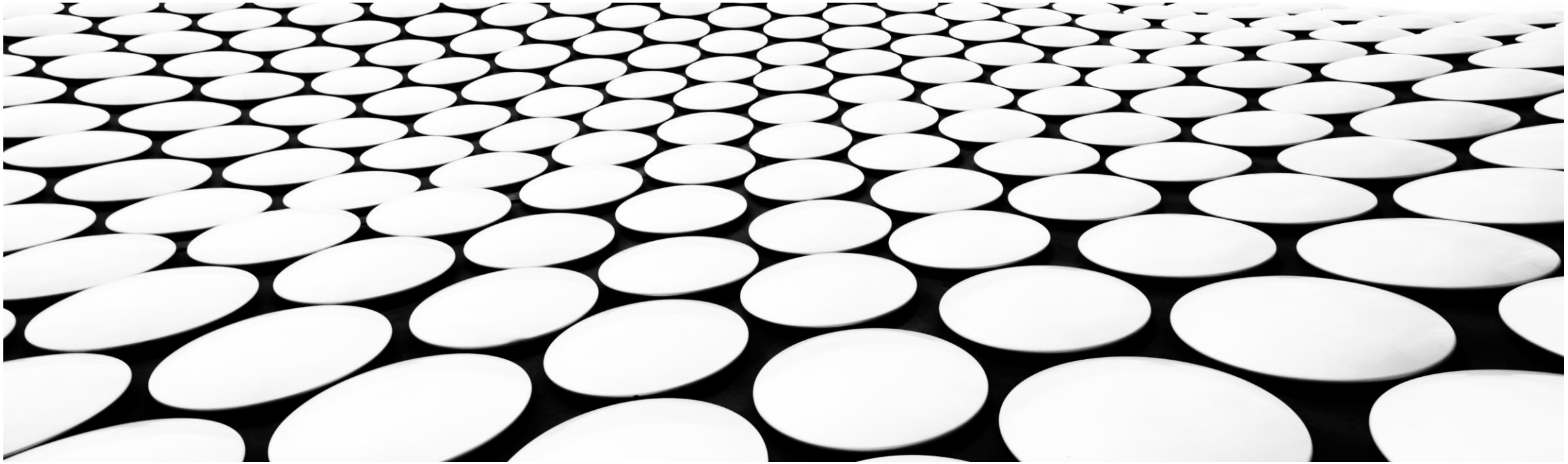
分布式计算

邱怡轩



今天的主题

- Python 函数式编程简介

前面的话

如何学习

- 编程枯不枯燥？
- 可以很枯燥（如果方法不当）

如何学习

- 我的一些建议

- 先明确目标（数据分析/当黑客/纯粹上头）
- 挑有用的部分先学（向量矩阵操作、字符串处理）
- 学会查找资料（搜索引擎、官方文档）

如何学习

- 在很多情况下，难的不是编程本身
- 而是解决问题的思路

- 对于大部分统计计算问题，当你能清晰地写出算法流程
- 编程只是把公式进行了一次“翻译”而已

本课程

- 本课程主要使用 Python 编程
- 几个层次

- 基本语法（变量、函数、条件、循环等等）
- 常见数据结构（列表、向量、矩阵等）
- 思维方式（函数式编程）

基本语法

- 都能学会，只是熟练与否
- 资料非常多，但没有必要“精读”
- 快速查找语法，如 https://www.runoob.com/python3/python3-tutorial.html

函数式编程

函数式编程

- 一种编程范式、风格、思维方式
- 与大数据处理、Spark 框架高度相关
- Python 提供了多种工具来实现
- 官方文档：
  https://docs.python.org/zh-cn/3/howto/functional.html

（比较枯燥，建议先看例子再读）

动机

- 你熟悉的数据处理方式：
  - 将数据一次性读取到内存
  - 从内存中抽取需要的数据进行操作
  - 内存占用大，操作灵活（数据按需获取）
- 函数式编程的处理方式：
  - 将数据表达为某种集合
  - 遍历集合中的元素，同时进行处理
  - 内存占用可控，对算法提出要求（遍历只能向前不能后退）

核心思想

- 将数据想像成信息流
- 每次只能看一小批数据
- 看完之后决定要做什么
- 操作完成后该批数据就被“丢弃”
- 下一批数据进入
- 往复循环

迭代器

迭代器

- 函数式编程中的核心概念
- 访问数据的一种机制

- 迭代器每次返回数据流中的一个元素
- 使用者不需要知道数据是如何存储的
- 只需要定义好对元素进行何种操作
- 迭代器只能向前不能后退
- YOLO: You Only Live/Look Once

迭代器

- 与传统方式的简单对比
- `lec3-functional.ipynb`（例1）

例子+练习

- 简单：计算文件行数
- 中等：计算序列样本方差
- 困难：随机抽样

练习1

- 给定一个文本文件，计算文件的行数
- 此时迭代器每次取出文件的一行，形成一个字符串
- `lec3-functional.ipynb`（例2）

练习2

- 给定一个数值型序列，计算其样本方差，要求只能遍历数据一遍
- `lec3-functional.ipynb`（例3）

练习3

- 给定一个正数序列 $(w_1, w_2, \ldots, w_n)$ 和一个数值序列 $(v_1, v_2, \ldots, v_n)$，要求从 $\{v_i\}$ 中随机抽取一个元素，其中每个 v_i 以 $p_i = w_i / \sum_j^n w_j$ 的概率被抽取到。要求只能遍历数据一遍
- `lec3-functional.ipynb` (例4)

练习3

Algorithm 1 The SELECT Algorithm.

Input: $\{a_1, \ldots, a_n\}$, $a_i \geq 0$, read in one pass, i.e., one sequential read, over the data.
Output: i^*, a_{i^*}.

1: $D = 0$.
2: **for** $i = 1$ to n **do**
3: $\quad D = D + a_i$.
4: $\quad$ With probability a_i/D, let $i^* = i$ and $a_{i^*} = a_i$.
5: **end for**
6: Return i^*, a_{i^*}.

The following lemma establishes that in one pass over the data one can sample an element according to certain probability distributions.

Lemma 1 *Suppose that* $\{a_1, \ldots, a_n\}$, $a_i \geq 0$, *are read in one pass, i.e., one sequential read over the data, by the* SELECT *algorithm. Then the* SELECT *algorithm requires* $O(1)$ *additional storage space and returns* i^* *such that* $\mathbf{Pr}[i^* = i] = a_i / \sum_{i'=1}^{n} a_{i'}$.

收集

- 如果想按传统的方式将迭代器的元素一次性取出来，可以将迭代器转换成列表
- `lst = list(it)`
- 但对于大型的数据，谨慎进行该操作
- 因为可能会占用非常多的内存

归约

- 即 Reduce 操作
- 将迭代器中的元素逐个进行二元运算
- `lec3-functional.ipynb`（例5）

迭代器变换

变换

- 在上节课中，我们使用了 PySpark 的 `filter()` 和 `map()` 来对数据进行处理
- 事实上，这正是函数式编程的思想

Filter

- Filter 将一个迭代器转变为一个新的迭代器
- 相当于对原始信息流进行了一次过滤（也是 Filter 这个函数名的由来）
- 只有满足条件的元素会被新迭代器访问
- `lec3-functional.ipynb`（例6）

Filter

```python
# Filter
def is_even(x):
    return x % 2 == 0


x = list(range(10))
# it 是原始的迭代器
it = iter(x)
print(next(it))  # 0
print(next(it))  # 1
print(next(it))  # 2
# it_filtered 是“过滤”后的迭代器
it = iter(x)
it_filtered = filter(is_even, it)
print(next(it_filtered))  # 0
print(next(it_filtered))  # 2
print(next(it_filtered))  # 4
```

Filter

- Filter 生成新的迭代器时不会实际进行过滤的计算
- 上例中 `it_filtered = filter(is_even, it)` 不会对原始数据进行计算
- 只有需要 `it_filtered` 返回元素时才会

Map

- Map 同样是将一个迭代器变换成另一个
- 新迭代器取出的元素都是原迭代器元素的变换
- 同样，生成迭代器时不会实际发生计算
- `lec3-functional.ipynb`（例7）

Map

```python
# Map
def square(x):
    return x * x


x = list(range(10))
# it 是原始的迭代器
it = iter(x)
print(next(it))  # 0
print(next(it))  # 1
print(next(it))  # 2
# it_mapped 是“变换”后的迭代器
it = iter(x)
it_mapped = map(square, it)
print(next(it_mapped))  # 0
print(next(it_mapped))  # 1
print(next(it_mapped))  # 4
```

其他工具

- `itertools` 模块提供了很多其他有用的迭代工具
- 例如 `itertools.islice()` 可以用来实现 PySpark 中的 `take()` 操作
- 生成一个截断长度的迭代器
- `lec3-functional.ipynb`（例8）
- 更多可参考https://docs.python.org/zh-cn/3/library/itertools.html#module-itertools

组合

- `map`, `filter`, `islice` 等可以互相叠加
- 本质上是把一个迭代器转换成另一个迭代器
- `lec3-functional.ipynb` (例9)

Lambda

- Map 和 Filter 都需要一个函数作为参数
- 通常可以先定义函数，再传入
- 而利用 Lambda 表达式（即匿名函数）可以在行内直接定义简单的函数
- https://www.runoob.com/python3/python3-function.html