STAT243 Lecture 5.7 Memory and Copies

1 Memory and Copies Overview

1.1 Overview

- 需要记住的两点：
  1. **数值数组（numeric arrays）** 每个元素占用 **8 bytes**；
  2. 需要留意何时创建了大型对象的**副本**（copy），以及何时只是复制了对象的**引用（reference）**。

Python

```python
x = np.random.normal(size=5)
x.itemsize  # 每个元素8字节
# 8
x.nbytes    # 总字节数 = 8 * 5 = 40
# 40
```

Logic

若数组包含 `n` 个元素且每个元素占 8 字节，则总内存大小约为 `8n bytes`。
对于大型 `NumPy` 数据结构，这一点尤其关键。

1.2 Allocating and Freeing Memory

- **Python 不需要显式内存分配**：
  与 C 这类编译语言不同，Python 会自动管理内存。
  当对象不再被引用时，系统通过 **垃圾回收（garbage collection）** 释放内存。
- **删除对象**
  - `del x`：仅解除变量名与对象之间的绑定；
  - 实际内存回收由垃圾回收器自动完成。
  - 一般无需手动调用 `gc.collect()`，除非要立即回收大对象占用的空间。

Remark

在 Python 中调用 `del` 并不会立刻释放内存，只是让该对象的引用计数归零。
当没有变量引用该对象时，Python 才会在下一次垃圾回收中释放它。

- **与 C 的比较**
  - 在 C 中，开发者必须手动分配 (`malloc`) 与释放 (`free`) 内存。
  - 若循环中忘记释放，会导致 **内存泄漏（memory leak）**。
  - Python 自动回收内存，**通常不会发生内存泄漏**，但当存在循环引用或外部资源时仍可能出现。

Logic

Python 的内存管理基于“引用计数（reference counting）+ 垃圾回收器（GC）”。
当对象的引用计数降为零时，其占用的内存会被标记为可回收。

1.3 The Heap and the Stack

- **堆（Heap）**
  - 程序运行时用于动态创建对象的内存区域。
  - 在 Python 中，大部分对象（包括列表、字典、NumPy 数组）都分配在堆上。
  - 当对象被删除或失去引用时，Python 负责将其内存释放回堆。
- **栈（Stack）**
  - 用于存储函数调用时的局部变量与执行帧（frames）。
  - 每次函数调用时会在栈上分配新的**栈帧（stack frame）**，函数结束后自动弹出。
  - “**Stack trace（调用栈追踪）**”即指当前所有活动函数帧的堆叠结构。

⚠ Remark

- 栈是**函数调用级**的内存结构，生命周期短且自动管理。
- 堆是**对象级**的内存结构，生命周期可跨函数存在。

因此：

- 在递归过深时会出现 **StackOverflowError**；
- 而在频繁创建大型对象时可能出现 **heap memory exhaustion**（堆内存不足）。

Logic

- **Stack**：临时变量、函数调用上下文（自动管理）。
- **Heap**：对象与动态数据（垃圾回收管理）。

它们的区别类似于短期与长期存储：
栈就像 CPU 的“笔记本”，堆更像电脑内存中长期保存的数据区。

2 Monitoring Memory Use

2.1 Monitoring Overall Memory Use on UNIX Systems

- **理解操作系统的内存缓存（Disk Caching）**
  - 操作系统会将从磁盘读取的文件或数据 **缓存（cache）到内存中**。
  - 若下次访问相同数据时仍在内存中，则无需重新从磁盘读取，大幅提高访问速度。
  - 尽管缓存会占用内存，但**这些缓存空间在需要时可立即被其他进程使用**，因此它是“可用”的。

Logic

“被缓存” ≠ “被占用”。
在 Linux 中，**缓存与缓冲（buff/cache）** 表示“可立即回收”的内存。
因此，系统内存即便显示“使用率高”，也可能仍有大量可用内存。

2.1.1 Using `free -h --si`

```Bash
free -h --si
```

选项说明：

- `-h` → human-readable，以更友好的单位显示（如 GB）。
- `--si` → 使用 **十进制 (GB)** 而非 **二进制 (GiB)** 单位。

示例输出：

```
               total        used        free      shared  buff/cache   available
Mem:           135G         24G         30G        7.8M         80G        110G
Swap:          274G         44G        230G
```

- `used`：当前被进程实际占用的内存（24G）
- `buff/cache`：用于缓存但可被回收的内存（80G）
- `available`：可立即供新进程使用的总量（≈ free + buff/cache = 110G）

2.1.2 Using `top`

```Bash
top
```

输出示例：

```
MiB Mem : 128877.0 total,  28825.9 free,  23856.4 used,  77249.1 buff/cache
MiB Swap: 262144.0 total, 220103.3 free,  42040.7 used. 105020.6 avail Mem
```

解释：

- 系统总内存：129 GiB
- 实际使用：24 GiB
- 可用内存：106 GiB（≈ 29 GiB free + 77 GiB buff/cache）

2.1.3 Swap 的含义

- **Swap 是磁盘上模拟的“虚拟内存”**。
- 当物理内存不足时，系统会将部分数据转移到 swap 空间。
- 由于磁盘读写速度远慢于内存，**使用 swap 会严重降低性能**。

> ⚠ **Remark**
>
> 如果 `Swap` 使用率很高（如上例 42 GiB），
> 即使还有充足的物理内存，也可能是系统内存管理策略或 I/O 优化造成。
> 通常，应尽量避免任务依赖 swap 空间。

> **Logic**

`free` 与 `top` 输出可能略有差异，
这可能源自单位差异（GB vs GiB）或系统更新周期不同。
不影响总体判断：只需关注 *Mem* 行的 used 与 available。

2.2 Monitoring Memory Use in Python

- **使用 UNIX 工具**
  - 在 Python 程序运行时，可直接在终端使用 `top` 或 `htop` 监控实时内存变化。

2.2.1 使用 `psutil` 获取进程内存

Python

```python
import psutil
memory_info = psutil.Process().memory_info()
print("Memory usage:", memory_info.rss / 10**6, "MB.")
# Memory usage: 310.6816 MB.
```

可封装为函数：

Python

```python
def mem_used():
    print("Memory usage:", psutil.Process().memory_info().rss / 10**6, " MB.")
```

Logic

`rss`（Resident Set Size）表示当前进程在物理内存中的占用量。

2.2.2 查看对象大小

- **`sys.getsizeof()`** 返回对象自身所占字节数：

  Python

  ```python
  my_list = [1, 2, 3, 4, 5]
  sys.getsizeof(my_list)   # 104 bytes
  x = np.random.normal(size=10**7)
  sys.getsizeof(x)         # ~80 MB (80000112 bytes)
  ```

- 但若对象**引用了其他对象**，结果可能低估内存占用：

  Python

  ```python
  y = [3, x]
  sys.getsizeof(y)   # 仅72 bytes!
  ```

Remark

`sys.getsizeof()` **不会递归计算引用对象的大小**。
若对象内部包含大型数组或字典，需使用更深入的测量方法。

2.2.3 更准确的测量：序列化法

Python

```python
import pickle
ser_object = pickle.dumps(y)
sys.getsizeof(ser_object)  # 80000202 bytes (~80 MB)
```

通过序列化（`pickle.dumps`）获取对象完整的二进制表示长度，可更准确反映真实内存占用。

2.2.4 其他工具与方法

- 启动 Python 时可添加调试选项以查看内存分配（详见 `man python`）。
- 专业分析工具：
  - `memory_profiler` – 逐行监测函数内存使用。
  - `pympler` – 提供详细对象级内存报告与可视化支持。

Logic

当进行大规模数据分析或机器学习时，
结合 `psutil` + `memory_profiler` 能有效识别内存瓶颈。

3 How Memory Is Used in Python

3.1 Two Key Tools: `id()` and `is`

- **`id(obj)`**
  返回对象在内存中的唯一标识（通常是对象的内存地址）。
- **`is`**
  判断两个变量是否指向同一个内存对象。

Python

```python
x = np.random.normal(size=10**7)
id(x)                    # 127279918838576
sys.getsizeof(x)         # 80000112 bytes

y = x
id(y)                    # 127279918838576
x is y                   # True
```

此时 `x` 与 `y` 共享同一内存。

Python

```python
z = x.copy()
id(z)                    # 不同内存地址
x is z                   # False
```

Logic

`id()` 用于追踪对象身份，而 `is` 判断“对象是否为同一个”。
若两个对象 `is` 相同，则修改其中一个会影响另一个。

3.2 Memory Use in Specific Circumstances

3.2.1 How Lists Are Stored

- 列表（list）是 **对象引用的集合**，而非一整块连续的内存。
- 每个元素是指向实际对象的 **指针（reference）**。

Python

```python
nums = np.random.normal(size=5)
obj = [nums, nums, np.random.normal(size=5), ['adfs']]
```

观察内存地址：

Python

```python
id(obj)          # 列表自身地址
id(obj[0])       # 指向 nums
id(obj[1])       # 同 nums
id(obj[2])       # 不同的数组
id(obj[3])       # 子列表对象
```

结果：

- `obj[0]` 与 `obj[1]` 指向同一个对象；
- 不同元素可引用相同对象。

Remark

列表本身仅存储引用（指针）。
访问 `obj[0]` 实际上执行一次索引取值并创建临时引用传递给 `id()`。

3.2.2 How Character Strings Are Stored

字符串（以及整数）在底层也可能被重用（interning），即相同文本值可能共用内存地址。
这些机制与 Python 的内存优化策略有关。

3.2.3 How NumPy Arrays Are Stored

- NumPy 数组的数据存储在 **连续内存块（contiguous memory block）** 中。
- 每个元素不是独立对象，而是 **8 字节浮点值** 的一部分。
- 访问元素时，会创建一个新的临时 Python `float64` 对象。

Python

```python
x = np.random.normal(size=5)
type(x[1])  # numpy.float64
```

```python
id(x[1])    # 每次都不同!
```

Logic

因为每次 `x[1]` 都会生成一个新的 Python 对象，
所以连续两次访问同一个元素，其 `id()` 不相同。

- 若将数组序列化（pickle），大小 ≈ 仅包含数值存储所需的空间 + 112 字节元数据。
- 若序列化一个包含相同数字的列表，则空间翻倍（因需存储引用指针）。

3.3 Modifying Elements In-Place

Python

```python
x = np.random.normal(size=5)
id(x)
x[2] = 3.5
id(x)
```

`id()` 不变，说明修改数组元素不会创建副本。

Logic

若每次修改都复制整个对象，
操作大型数组几乎不可行，因此 NumPy 允许原地修改。

3.4 Shallow Copying

- **浅拷贝（shallow copy）**：仅复制最外层容器，内部元素仍指向原对象。

Python

```python
x = [3, [1,2,3]]
y = x.copy()
x[0] = 9
y[1][2] = 99
x  # [9, [1, 2, 99]]
```

Remark

改变嵌套元素（如子列表）会影响原对象，
因为它们仍共享相同的引用。

- **深拷贝（deep copy）**：复制整个结构，不共享任何子对象。

Python

```python
import copy
x = [3, [1,2,3]]
y = copy.deepcopy(x)
y[1][2] = 99
x  # [3, [1, 2, 3]]
```

3.5 When Are Copies Made?

Python

```python
x = np.random.normal(size=10**8)
y = x            # 同一对象，共享内存
x = np.random.normal(size=10**8)  # 新分配内存
```

Logic

只有当变量重新绑定（reassignment）到新对象时，
Python 才会分配新内存。

3.6 How Does Python Know When to Free Memory?

- Python 采用 **引用计数（reference counting）**。
- 每个对象都有一个引用计数，当计数为 0 时即被销毁并回收内存。

Python

```python
import sys
x = np.random.normal(size=10**8)
y = x
sys.getrefcount(y)  # 3
del x
sys.getrefcount(y)  # 2
del y               # 对象被释放
```

Remark

`getrefcount()` 返回值比预期多 1，
因为函数调用本身会临时创建一个引用。

示例：

Python

```python
x = np.random.normal(size=5)
sys.getrefcount(x)  # 实际1，显示2
y = x
sys.getrefcount(x)  # 实际2，显示3
del y
sys.getrefcount(x)  # 实际1，显示2
```

Logic

这种机制类似于 C++ 的 **shared pointers** 或 R 的 **copy-on-write**。
当最后一个引用消失时，对象即被垃圾回收（GC）回收释放内存。

4 Strategies for Saving Memory

4.1 Basic Strategies

1. **Avoid unnecessary copies**
   - 不要轻易复制大型对象。
   - 优先使用引用（reference）而非 `.copy()`，除非确实需要独立副本。
2. **Remove unused objects**
   - 删除不再使用的变量：

     Python
     ```python
     del large_array
     ```

   - 这会解除名称与对象的绑定，让垃圾回收器（GC）自动回收内存。
3. **Use iterators and generators**
   - 避免创建大型中间列表，例如：

     Python
     ```python
     # 差：占用大量内存
     nums = [i**2 for i in range(10**7)]
     # 好：使用生成器，节省内存
     nums = (i**2 for i in range(10**7))
     ```

   - 迭代器（iterator）与生成器（generator）**惰性求值**，只在需要时生成元素。

> **Logic**
>
> Python 的列表会一次性将所有元素加载入内存，而生成器仅保存计算逻辑与当前位置索引。
> 因此，对于大规模数据循环处理，生成器能显著降低内存占用。

4.2 Advanced Optimization

- **使用更紧凑的数据类型**
  - 浮点型默认是 `float64`（8 bytes），但在精度要求不高时可用更小类型。

    Python
    ```python
    x = np.array(np.random.normal(size=5), dtype="float32")
    x.itemsize  # 4 bytes
    x = np.array([3,4,2,-2], dtype="int16")
    x.itemsize  # 2 bytes
    ```

  - 减小 `dtype` 直接减少数组内存占用。

> **Remark**
>
> 在数值分析中，使用 `float32` 可能会造成累积误差或舍入误差；
> 因此仅在确认结果可接受时才降精度。

- **分块（chunked）读取数据**
  - 对大文件，不必一次性加载全部数据，可采用分块读取：

    Python
    ```python
    for chunk in pd.read_csv("data.csv", chunksize=10000):
    ```

```
    process(chunk)
```

  - 有助于防止内存峰值溢出，尤其在处理数 GB 文件时。
- **使用高效的存储格式 / 工具包**
  - 可考虑使用 **Apache Arrow**、**Parquet** 等高效的内存结构与文件格式。
  - 它们支持零拷贝（zero-copy）与压缩编码，大幅减少内存占用。

4.3 Example: Memory Allocation in Practice

以下代码展示了一个示例函数 `fastcount()`，
它用于统计两个大型数组 `x` 与 `y` 中非 NaN 元素的组合，但会引发多次内存分配：

Python

```python
def fastcount(xvar, yvar):
    naline = np.isnan(xvar)
    naline[np.isnan(yvar)] = True
    localx = xvar.copy()
    localy = yvar.copy()
    localx[naline] = 0
    localy[naline] = 0
    useline = ~naline
    ## ...
```

⚠ Remark

内存分配点：

1. `np.isnan(xvar)` → 新建布尔数组（与 `xvar` 同尺寸）
2. `np.isnan(yvar)` → 另一个布尔数组
3. `xvar.copy()` 与 `yvar.copy()` → 两个完整副本（约两倍原内存）
4. `naline` 的逻辑运算（`~naline`）生成新数组 `useline`

由此可见，在处理大型 NumPy 数组时：

- 每次 `.copy()`、逻辑运算或布尔索引操作都会触发新的内存分配；
- 若输入数组在 1 GB 量级，整个函数可能瞬间占用 3–4 GB 内存。

Logic

优化建议：

- 尽量使用 **in-place 操作**（如 `np.nan_to_num(xvar, copy=False)`）；
- 复用已有布尔掩码数组而非重复创建；
- 若结果不需长期保存，及时释放局部变量以触发垃圾回收。