计算导论与 Python 编程

期末复习

李谨硕

2026 年 1 月 7 日

目录

1 Linux 及 Python 常用命令

在计算导论与实验课程中，掌握基础的命令行操作是进行编程开发的前提。本节涵盖了文件系统操作、系统管理、网络通信以及 Python 环境管理的核心命令。

1.1 Linux 文件与目录管理

文件操作是 Linux 交互中最基础的部分，考试中常考路径的相对与绝对引用以及文件权限的理解。

ls (List) 列出目录内容。

- `ls`：仅列出文件名。
- `ls -l`：长格式显示（包含权限、大小、时间）。
- `ls -a`：显示所有文件（包含以 . 开头的隐藏文件）。

cd 切换当前工作目录。

- `cd /path/to/dir`：跳转到绝对路径。
- `cd ..`：返回上一级目录。
- `cd ~`：返回当前用户的家目录（Home）。

mkdir 创建新目录。常用参数 `-p` 可递归创建多级目录（例如 `mkdir -p a/b/c`）。

touch 用于修改文件时间戳，若文件不存在则创建一个空文件。

cp 复制文件或目录。

```
# 将 file1 复制为 file2
cp file1.txt file2.txt
# 递归复制目录（注意 -r 参数）
cp -r source_folder/ destination_folder/

```

Listing 1: cp 命令示例

mv 移动文件或重命名文件。

- `mv old.txt new.txt`：重命名。
- `mv file.txt ./folder/`：移动文件。

rm 删除文件或目录。

[!] 考试指南：rm 命令的危险性

在使用 `rm` 时需格外小心，Linux 下删除通常无法恢复！

- `rm -r`：递归删除目录。
- `rm -f`：强制删除，不提示确认。
- **绝对禁忌**：`rm -rf /`（这将删除整个系统文件，实战中极度危险）。

1.2 系统管理与网络命令

此类命令用于查看系统状态、安装软件及处理网络请求。

ps 查看当前进程快照。

- `ps al`: 显示所有进程的详细信息。
- `ps aux`：显示所有用户的所有进程详细信息。
- 常配合 `grep` 使用查找特定进程（例如 `ps aux | grep python`）。

apt Debian/Ubuntu 系 Linux 的包管理器。

```
sudo apt update          # 更新软件源列表
sudo apt install git     # 安装 git 软件
sudo apt upgrade         # 升级所有已安装软件

```

ssh 安全远程登录协议。

- 语法：`ssh user@hostname`
- 示例：`ssh student@192.168.1.100`

ping 测试网络连通性。通过发送 ICMP 数据包检测目标主机是否可达。

wget 从网络下载文件。

- 示例：`wget http://example.com/file.zip`

1.3 Python 环境与包管理命令

1.3.1 python 命令

用于执行 Python 脚本或进入交互式环境。

- **查看版本：** `python --version` 或 `python -V`
- **运行脚本：** `python main.py`
- **模块运行：** `python -m http.server` (例如启动简易服务器)

1.3.2 pip 命令

Python 的包安装程序 (Package Installer for Python)。

命令	功能说明
`pip install numpy`	安装 numpy 包
`pip uninstall pandas`	卸载 pandas 包
`pip list`	列出已安装的所有包
`pip freeze > requirements.txt`	将当前环境包列表导出到文件
`pip install -r requirements.txt`	根据文件批量安装依赖

表 1: pip 常用指令速查

Pip 下载速度慢怎么办?

国内网络环境下，直接使用默认源下载速度可能较慢。可以使用 `-i` 参数指定国内镜像源（如清华源）：

```
pip install matplotlib -i https://pypi.tuna.tsinghua.edu.cn/simple
```

或者配置永久换源。

1.4 命令功能速查表

命令	全称/含义	核心功能
`ls`	List	列出当前目录下的文件
`cd`	Change Directory	切换目录
`pwd`	Print Working Directory	显示当前所在路径
`cp`	Copy	复制文件或文件夹（需加 -r）
`mv`	Move	移动或重命名文件
`rm`	Remove	删除文件（不可逆）
`mkdir`	Make Directory	创建文件夹
`touch`	Touch	创建空文件或更新时间戳

命令	全称/含义	核心功能
`cat`	Concatenate	查看文件内容
`ps`	Process Status	查看进程状态
`apt`	Advanced Package Tool	软件安装与管理
`ssh`	Secure Shell	远程登录
`ping`	Packet Internet Groper	测试网络连通性
`wget`	World Wide Web Get	命令行下载工具

2 Python 基本类型与运算

2.1 基本数据类型

Python 中的变量不需要声明类型，类型由赋值决定。

- **整数 (int)**: 任意大小的整数，如 `10`, `-5`。
- **浮点数 (float)**: 小数，如 `3.14`, `1.2e-5` (科学计数法)。
- **复数 (complex)**: 形如 `a + bj`，如 `3+4j`。实部 `.real`, 虚部 `.imag`。
- **字符串 (str)**: 单引号或双引号括起来，如 `'abc'`, `"Hello"`。
- **布尔值 (bool)**: 只有 `True` 和 `False` (注意首字母大写)。

2.2 类型转换函数

考试中常考强制类型转换的结果，特别是浮点转整数（截断）和进制转换。

函数	说明	示例 (Input → Output)
`int(x)`	转为整数 (截断小数)	`int(3.9)` → `3`
`float(x)`	转为浮点数	`float(5)` → `5.0`
`str(x)`	转为字符串	`str(123)` → `'123'`
`bool(x)`	非 0 非空即为真	`bool(0)` → `False`, `bool(2)` → `True`
`bin(x)`	转为二进制字符串	`bin(10)` → `'0b1010'`

2.3 算术运算符

常规运算 (+, -, *, +=, -=) 不再赘述，重点掌握以下特殊运算：

运算符	名称	示例与说明
/	真除法	5 / 2 → 2.5 (结果总是 float)
//	整除 (地板除)	5 // 2 → 2, -5 // 2 → -3 (向下取整)
%	取模 (求余)	5 % 2 → 1, 10 % 3 → 1
**	幂运算	2 ** 3 → 8, 9 ** 0.5 → 3.0 (开方)

[!] 考试指南：负数的地板除与取模

Python 的 // 是**向下取整**。

- -5 // 2 结果是 -3 (而不是 -2)。
- 取模公式：a % b = a - (a // b) * b。

2.4 逻辑与比较运算符

- **比较**: == (等于), != (不等于), <=, >=, <, >
- **逻辑**: and, or, not
- **成员**: in, not in (例如 'a' in 'apple' 返回 True)
- **身份**: is, is not (判断对象内存地址是否相同)

is 和 == 的区别

- == 判断**值 (Value)** 是否相等。
- is 判断**引用 (Reference)** 是否指向同一个对象 (即 ID 是否相同)。

例如：

```
a = [1, 2]
b = [1, 2]
print(a == b)  # True (内容一样)
print(a is b)  # False (是两个不同的列表对象)
```

2.5 位运算

针对二进制位的操作 (将数字视为二进制处理)。

符号	描述	计算示例 (设 a=5(101), b=3(011))
`&`	按位与	`5 & 3` → `1` (001)
`|`	按位或	`5 | 3` → `7` (111)
`^`	按位异或	`5 3` → `6` (110) (相同为 0，不同为 1)
`<<`	左移	`5 << 1` → `10` (1010) (相当于乘 2)
`>>`	右移	`5 >> 1` → `2` (010) (相当于整除 2)

2.6 range() 函数详解

`range` 用于生成整数序列，常用于 `for` 循环。** 注意：左闭右开区间 **（包含开始，不包含结束）。

`range(stop)` 从 0 开始，到 stop 结束（不含 stop）。

示例: `range(3)` → `0, 1, 2`

`range(start, stop)` 从 start 开始，到 stop 结束（不含 stop）。

示例: `range(2, 5)` → `2, 3, 4`

`range(start, stop, step)` 步长为 step。

示例: `range(1, 10, 2)` → `1, 3, 5, 7, 9`

倒序: `range(5, 0, -1)` → `5, 4, 3, 2, 1`

3 变元与赋值语句

理解 Python 的变量本质（引用语义）是掌握语言核心的关键。Python 中的变量更像是贴在对象上的**便利贴（标签）**，而不是装数据的盒子。

3.1 指向关系与赋值分类

- **变量 (Variable)**: 仅仅是一个名字（引用），指向内存中的对象。
- **对象 (Object)**: 存储实际数据的实体，拥有类型 (type)、值 (value) 和唯一标识 (id)。

重新赋值 (Re-assignment)	形式为 `a = ...`。 改变的是**变量的指向**，让变量指向一个新的对象。原对象如果引用计数为 0 会被回收。

修改赋值 (Mutation) 形式为 `a[i] = ...` 或 `a.x = ...`。
变量指向的**对象本身发生了改变**（内容更新），但变量依然指向该对象（ID 不变）。**前提是该对象支持修改（即 Mutable）**。

```
a = [1, 2, 3]    # a 指向列表对象 <Obj_1>
b = a            # b 也指向 <Obj_1>

# 1. 修改赋值 (Mutation)
a[0] = 99        # <Obj_1> 内容变为 [99, 2, 3]
print(b)         # [99, 2, 3] -> b 受到影响，因为指向同一个对象

# 2. 重新赋值 (Re-assignment)
a = [4, 5]       # a 撕掉标签，贴到了新对象 <Obj_2> 上
print(b)         # [99, 2, 3] -> b 依然指向 <Obj_1>，不受影响
```

Listing 2: 重新赋值 vs 修改赋值

3.2 不可变 (Immutable) vs 可变 (Mutable)

类型	常见数据结构	特点
不可变 (Immutable)	`int`, `float`, `str`, `tuple`, `bool`	一旦创建，内容不可改。
可变 (Mutable)	`list`, `dict`, `set`	内容可以就地修改，ID 保持不变。

3.3 对象创建与复用规则

3.3.1 1. 赋值语句中的对象创建

原则上，赋值语句右侧的表达式每次运算都会创建一个**新对象**。

- **可变对象不复用**：即使内容相同，每次创建的 List/Dict 都是独立的。
- **变量传递无拷贝**：如果表达式只是单个变量 `b = a`，则**不创建新对象，也不进行拷贝**，仅仅是传递引用（多了一个标签）。

```
x = [1, 2]
y = [1, 2]
print(x == y)  # True (值相等)
print(x is y)  # False (不是同一个对象, 内存地址不同)

z = x
print(z is x)  # True (直接指向, 不产生拷贝)
```

3.3.2 2. 不可变对象的特殊共享 (Interning)

为了优化性能，Python 对**小整数**（通常是 -5 到 256）和**短字符串**会进行缓存复用。

```
a = 100; b = 100
print(a is b)  # True (触发了小整数缓存机制)
```

3.4 重点难点：+= 的区别

+= (In-place Add) 对于可变和不可变对象有截然不同的行为，这是考试的**必考坑点**。

[!] 考试指南：+= 操作符的陷阱

- **对于 Immutable (如 int, str)：**
  `a += b` 等价于 `a = a + b`。
  效果：创建新对象，重新赋值。变量 `a` 的 ID 会改变。
- **对于 Mutable (如 list)：**
  `a += b` 等价于 `a.extend(b)`。
  效果：就地修改原对象。变量 `a` 的 ID **保持不变**。

经典考题示例：

```
# --- Case 1: List (Mutable) ---
a = [1, 2]
b = a
a += [3]        # 在原列表追加，a 依然指向原对象
print(b)        # 输出 [1, 2, 3] (b 被"连累"了)

# --- Case 2: Int (Immutable) ---
x = 10
y = x
x += 1          # 计算 11，x 指向新对象 11
print(y)        # 输出 10 (y 依然指向旧对象 10)
```

4 条件分支

4.1 判定标准：is 与 ==

在 `if` 判断中，必须分清“值相等”与“身份相同”。

- `==`：判断**值 (Value)** 是否相等（调用 `__eq__`）。绝大多数业务逻辑使用此符号。
- `is`：判断**身份 (Identity)** 是否相同（即内存地址是否一致，`id(a) == id(b)`）。通常用于判断 `None`。

```python
a = [1, 2]
b = [1, 2]
if a == b:
    print("值相等")       # 会输出
if a is b:
    print("是同一个对象") # 不会输出（两个独立的列表）
```

Listing 3: is 与 == 的对比

4.2 真值判定 (Truthiness)

Python 中不仅仅是 `True/False` 可以作为条件，**任何对象**都可以放在 `if` 后面进行判定。

[!] 考试指南：什么会被视为 False?

除了明确的 `False`，以下值在布尔上下文中也被视为 **假 (False)**：

- **None**
- **数字零**: `0`, `0.0`, `0j`
- **空序列**: `''`, `""` (空字符串), `[]` (空列表), `()` (空元组)
- **空集合**: `{}` (空字典), `set()`

除上述情况外，其他所有值通常都视为 True。

示例：

```python
name = "Alice"
if name:           # 字符串非空，判定为 True
    print("Hi")

mylist = []
if mylist:         # 列表为空，判定为 False，不执行
    print("Full")
```

4.3 基本语法结构

4.3.1 1. if-elif-else 结构

最基础的逻辑控制。注意缩进必须一致。

```
score = 85
if score >= 90:
    print("A")
elif score >= 80:  # 可以有多个 elif
    print("B")
else:              # 可选
    print("C")
```

4.3.2 2. 结构化模式匹配 (Match-Case)

(Python 3.10+ 新特性，类似 C 语言的 switch，但更强大)

```
status = 404
match status:
    case 200:
        print("Success")
    case 404:
        print("Not Found")
    case _:        # 相当于 default/else，匹配所有剩余情况
        print("Unknown error")
```

4.4 条件表达式 (三元运算符)

Python 中没有 `expression ? true : false` 语法，而是使用更像自然语言的结构。

语法格式

变量 = 值 1 if 条件 else 值 2

示例：

```
age = 20
# 如果 age > 18，status 为 'Adult'，否则为 'Teen'
status = "Adult" if age > 18 else "Teen"
```

5 循环结构

5.1 基础循环与控制

for 循环 用于遍历可迭代对象（如 List, String, Dict, Range）。

while 循环 当条件满足（True）时重复执行，适合不知道具体循环次数的场景。

break **跳出整个循环**。不再执行循环体内的后续代码，也不再进行后续迭代。

continue **跳出当次迭代**。忽略本次循环体剩余代码，直接开始下一次迭代。

5.2 循环中的 else 子句

Python 的 `for` 和 `while` 都可以搭配 `else` 使用。这是考试中的**逻辑陷阱**。

else 什么时候执行?

执行原则：只有当循环**正常结束**（即没有被 `break` 打断）时，`else` 块才会执行。

```python
# --- 情况 1: 遇到 break ---
for i in range(5):
    if i == 3:
        break        # 强制退出
else:
    print("Done")    # 不会被执行!

# --- 情况 2: 正常跑完 ---
for i in range(5):
    pass
else:
    print("Done")    # 会被执行
```

Listing 4: break 与 else 的互斥关系

5.3 循环中修改容器 (重点难点)

在遍历列表或字典时修改它们，是极易出错的操作。需区分“修改值”和“修改大小”。

5.3.1 1. 修改 Value (安全)

如果容器的**长度 (Size) 保持不变**，仅修改内容是允许的。

```python
nums = [1, 2, 3]
for i in range(len(nums)):
    nums[i] = nums[i] * 2    # Safe: 长度没变，只是改了值
```

5.3.2 2. 修改 Size (危险 - 添加/删除元素)

[!] 考试指南：RuntimeError: dictionary changed size

在 for 循环遍历 **Dict** 或 **Set** 时，如果**增加**或**删除**键值对，Python 会直接抛出 RuntimeError。

注：*List* 虽然不会直接报错，但会导致索引混乱（漏删或越界），同样危险。

错误示范：

```python
d = {'a': 1, 'b': 2}
for k in d:
    if k == 'a':
        del d[k]  # 报错! RuntimeError
```

正确解法：使用副本 (.copy()) 如果必须在循环中修改大小，请遍历容器的**副本**。

```python
# 方案 A: 使用 .copy() (推荐)
d = {'a': 1, 'b': 2}
for k in d.copy():  # 遍历的是复制出来的 keys
    if k == 'a':
        del d[k]    # 修改的是原字典 d

# 方案 B: 使用 while 循环 (适合 List)
nums = [1, 2, 3, 4]
i = 0
while i < len(nums):
    if nums[i] % 2 == 0:
        del nums[i] # 删除后不移动指针
    else:
        i += 1      # 只有不删除时才移动指针
```

Listing 5: 安全删除元素的写法

6 函数 (Functions)

6.1 定义与调用基础

- **定义**: 使用 `def` 关键字。
- **Return**:
  - `return` 语句执行后，函数**立即终止**。
  - 如果函数体执行完毕没有遇到 `return`，或者写了 `return` 但后面没跟值，默认返回 `None`。
- **对象方法调用**: 语法为 `obj.method(args)`，例如 `lst.count(1)`。这实际上是将 `obj` 作为隐含的第一个参数传递给函数。

6.2 参数类型与规则

Python 的参数处理非常灵活，需严格掌握以下四种类型及其顺序。

6.2.1 1. 参数分类

Positional Args 位置参数。按顺序对应。

Keyword Args 关键字参数。调用时指定 `name=value`。

Arbitrary Posi. `*args`。接收多余的位置参数，打包成 **Tuple**。

Arbitrary Key. `**kwargs`。接收多余的关键字参数，打包成 **Dict**。

缺省参数 (Default Arguments) 的限制

1. **位置限制**：在函数定义时，**缺省参数必须放在非缺省参数之后**。
   错误示例: `def f(a=1, b): ...` (SyntaxError)
   正确示例: `def f(b, a=1): ...`
2. **调用限制**：一旦在调用中使用了关键字参数，其后的所有参数都必须使用关键字形式。

6.3 参数传递机制 (核心考点)

Python 的参数传递既不是纯粹的“传值”，也不是“传引用”，而是**“传对象引用 (Call by Object Reference)”**。即传入的是对象的**内存地址**。

[!] 考试指南：重新赋值 vs 修改对象

函数内部的操作是否影响外部变量，取决于操作类型：

- **重新赋值 (Re-assignment) - 不影响外部**
  `x = ...` 只是让局部变量 x 指向了新对象，原外部对象不动。
- **修改对象 (Mutation) - 影响外部**
  `x[0] = ...`, `x.append()`, `x.attr = ...` 是顺着地址修改了**原对象**的内容。

```python
def modify(a_list, b_list):
    # 1. 重新赋值 (Re-assignment)
    a_list = [99, 99]       # a_list 贴到了新列表上，断开了与外部的联系

    # 2. 修改内容 (Mutation)
    b_list.append(100)      # 顺着地址修改了外部传入的那个列表对象

x = [1, 2]
y = [1, 2]
modify(x, y)

print(x)  # [1, 2]        -> 未变
print(y)  # [1, 2, 100]  -> 已变
```

Listing 6: 修改外部对象的对比

6.4 高阶特性：嵌套、装饰器与 Lambda

6.4.1 1. 嵌套函数 (Nested Functions)

函数内部可以定义函数，也可以将内部函数作为返回值返回（闭包的基础）。

6.4.2 2. 装饰器 (Decorator)

本质上是一个接收函数作为参数并返回新函数的高阶函数。

```python
@my_decorator
def my_func():
    pass
# 等价于: my_func = my_decorator(my_func)
```

6.4.3 3. Lambda 表达式

匿名函数，通常用于简单的单行逻辑。

- **语法**: `lambda arguments: expression`
- **注意**: 只能包含一个表达式，不能包含复杂的语句（如赋值、循环）。隐式返回表达式的结果。

```
f = lambda x, y: x + y
print(f(1, 2))  # 输出 3

# 常用于排序
pairs = [(1, 'one'), (3, 'three'), (2, 'two')]
pairs.sort(key=lambda p: p[1]) # 按第二个元素排序
```

6.5 递归 (Recursion)

函数调用自身。

- 必须有**基准情况 (Base Case)** 以结束递归，否则会导致栈溢出 (RecursionError)。
- 每次递归调用都会在内存栈中开辟新的帧 (Frame)，保存当次调用的局部变量。

7 常用内置函数

Python 提供了大量开箱即用的内置函数，考试中需重点掌握其参数行为及返回值类型。

7.1 输入输出与动态执行

print(*objects) 输出对象到标准输出。

- `sep`: 多个对象之间的分隔符，默认为空格。
- `end`: 输出结束后的字符，默认为换行符。

```
print("A", "B", sep="-", end="!")
# 输出: A-B! (而不是 A B\n)

```

eval 将字符串作为 Python 表达式执行，并返回结果。
功能强大：可以处理复杂的字面量结构。

```python
s = "[1, 2, 3]"
lst = eval(s)    # 将字符串转换为列表对象
print(lst[0])    # 输出 1

```

7.2 对象自省 (Introspection)

type(obj) 返回对象的类型对象。

id(obj) 返回对象的唯一标识符（通常是内存地址）。

len(obj) 返回容器（字符串、列表、字典等）的元素个数。

isinstance 判断 obj 是否是 class 的实例（**支持继承关系判定**）。

issubclass 判断 cls 是否是 class 的子类。

type() vs isinstance()

考试中常问两者的区别：

- `type(x) == A`：**不考虑继承**。如果 x 是 A 的子类实例，结果为 False。
- `isinstance(x, A)`：**考虑继承**。如果 x 是 A 的子类实例，结果为 True。（推荐使用）

7.3 数学运算与序列操作

sum 对序列进行求和。注意可指定起始值 `start`。

max/min 返回最大值/最小值。
关键参数 key: 类似于 sort，可以指定排序依据。
`max(["a", "abc"], key=len)` → `"abc"`

7.4 函数式编程与排序 (高频考点)

sorted 返回一个新的**已排序列表**。
区别: `list.sort()` 是就地修改，返回 None；`sorted()` 返回新对象，原对象不变。

map 将 func 作用于 iterable 的每一个元素。

filter 将 iterable 中使得 `func(x)` 为 True 的元素保留下来。

[!] 考试指南：Python 3 中的 map 和 filter

在 Python 3 中，`map()` 和 `filter()` 返回的不再是列表，而是**迭代器 (Iterator)**。它们是**惰性求值 (Lazy)** 的，只有在遍历或转换为 list 时才会计算。

```
m = map(str, [1, 2, 3])
print(m)         # <map object at ...> (不会直接打印内容)
print(list(m))   # ['1', '2', '3'] (转换后可见)
```

8 列表 (List) - 核心可变序列

列表是 Python 中最常用的**可变 (Mutable)** 序列类型。

8.1 基本操作与构造

- **表示**: 使用方括号 `[]`, 元素间用逗号分隔。
- **构造函数**: `list(iterable)`。可将元组、字符串、range 等转换为列表。
- **运算**:
  - **加法 (+)**: `[1, 2] + [3]` → `[1, 2, 3]` (连接，产生新列表)。
  - **乘法 (*)**: `[1] * 3` → `[1, 1, 1]` (重复，产生新列表)。
- **成员判定**: `x in lst` (时间复杂度 O(n))。

8.2 索引与切片 (Indexing & Slicing)

8.2.1 1. 访问 (Access)

- **Index**: `a[i]`。支持负数索引（-1 为最后一个）。
- **Slice**: `a[start:stop:step]`。
  缺省值: `start` 默认为 0，`stop` 默认为长度，`step` 默认为 1。
  常用技巧: `a[::-1]` (反转列表), `a[:]` (浅拷贝整个列表)。

8.2.2 2. 修改 (Modification) - 重点

列表是可变的，支持原位修改。

```
lst = [0, 1, 2, 3, 4]

# 索引修改 (必须存在)
lst[0] = 99          # -> [99, 1, 2, 3, 4]

# 切片修改 (功能强大: 可替换、插入、删除)
# 将索引 1 到 3 的片段替换为新的列表
lst[1:3] = ['a', 'b', 'c']
# 结果: [99, 'a', 'b', 'c', 3, 4] (长度可以改变)

# 即使是切片赋值, 右侧也必须是 Iterable
lst[1:2] = [100]     # 正确
# lst[1:2] = 100     # 报错! TypeError
```

Listing 7: 索引修改 vs 切片修改

8.3 常用方法 (Methods) 与删除

方法/操作	说明
`.append(x)`	在末尾添加元素 x。
`.extend(iterable)`	将 iterable 中的所有元素追加到末尾。
`.insert(i, x)`	在索引 i 处插入 x。
`.remove(x)`	删除**第一个**值为 x 的元素。如果不存在则报错。
`.pop([i])`	删除并返回索引 i 处的元素（默认最后一个）。
`.clear()`	清空列表，变为 []。
`.index(x)`	返回第一个 x 的索引。
`.count(x)`	统计 x 出现的次数。
`.sort(key=.., reverse=..)`	**就地排序 (In-place)**，无返回值 (None)。
`.reverse()`	**就地反转**，无返回值。
`del lst[i]`	删除指定索引的元素。
`del lst[start:stop]`	删除切片范围内的元素。

8.4 列表推导式 (List Comprehension)

一种简洁构建列表的方法，通常比 for 循环更快。

- **基础语法**: `[expression for item in iterable if condition]`
- **嵌套 (Nested)**: 对应嵌套的 for 循环。

```
# 目标: 展平二维数组 matrix = [[1, 2], [3, 4]]
flat = [num for row in matrix for num in row]
# 等价于:
# for row in matrix:
#     for num in row: ...
```

Listing 8: 嵌套推导式示例

8.5 核心难点：拷贝机制 (Shallow vs Deep Copy)

这是列表最容易出错的地方，必须严格区分三种拷贝层级。

8.5.1 1. 引用赋值 (Reference)

`b = a`。不创建新对象，`a` 和 `b` 指向同一块内存。

8.5.2 2. 浅拷贝 (Shallow Copy)

创建一个**新列表容器**，但列表里面的元素依然是**原对象的引用**。
触发方式:

- `b = a[:]` (切片)
- `b = a.copy()`
- `b = list(a)`

```
a = [[1], [2]]       # 嵌套列表
b = a.copy()         # 浅拷贝

b[0].append(9)       # 修改内部对象
print(a)             # [[1, 9], [2]] -> 原列表 a 也变了!
# 因为 b[0] 和 a[0] 指向的是同一个小列表对象
```

Listing 9: 浅拷贝的局限性

8.5.3 3. 深拷贝 (Deep Copy)

递归地拷贝列表及其包含的所有子对象。需导入 copy 模块。

```
import copy
c = copy.deepcopy(a) # 完全独立，修改 c 不会影响 a
```

8.5.4 4. 乘法操作符 * 的陷阱

lst * n 执行的是浅拷贝。

[!] 考试指南：千万别这样初始化二维数组！

错误写法：

```
# 创建一个 3x3 矩阵
matrix = [[0] * 3] * 3
# 结果：[[0,0,0], [0,0,0], [0,0,0]]
# 问题：这3个内部列表其实是同一个对象的引用！

matrix[0][0] = 99
print(matrix)
# 结果：[[99,0,0], [99,0,0], [99,0,0]] -> 3行全变了
```

正确写法 (List Comprehension)：

```
# 推导式每次循环都会重新计算表达式 -> 创建新的列表对象
matrix = [[0] * 3 for _ in range(3)]
```

List Comprehension 的求值特性

推导式中的表达式（expression）在每次迭代时都会**重新求值 (Re-evaluates)**。

- [[1] for _ in range(3)]: 循环 3 次，每次都创建一个新的 [1]。相当于对 [1] 做了 **Deep Copy** 的效果（虽然技术上不是通过 deepcopy 实现的）。

9 元组 (Tuple) - 不可变序列

元组可以被视为**不可变的列表 (Immutable List)**。一旦创建，其长度和内部元素的指向都不能改变。

9.1 表示与构造

- **符号**: 使用圆括号 `()`。
- **构造函数**: `tuple(iterable)`。
- **空元组**: `t = ()`。

[!] 考试指南：单元素元组的陷阱

创建只有一个元素的元组时，**必须在元素后加逗号**，否则 Python 会将其识别为普通的数学运算括号（即该元素本身的类型）。

```
t1 = (1)     # <class 'int'> (整数 1)
t2 = (1,)    # <class 'tuple'> (元组)
```

9.2 基本操作

元组支持大部分列表的**只读**操作：

- **索引 (Indexing)**: `t[0]`, `t[-1]`。
- **切片 (Slicing)**: `t[1:3]`。返回一个新的元组。
- **成员判定**: `x in t`。
- **运算**:
  - **加法**: `(1, 2) + (3,)` → `(1, 2, 3)` (创建新元组)。
  - **乘法**: `(1,) * 3` → `(1, 1, 1)`。

9.3 常用方法

由于不可变，元组**没有** `append`, `extend`, `remove` 等修改方法。仅支持查询：

- `.count(x)`: 统计 x 出现的次数。
- `.index(x)`: 返回 x 第一次出现的索引。

9.4 特殊考点：没有” 元组推导式”

Tuple Comprehension 不存在

在 Python 中，使用圆括号包裹的推导式语法 (`x for x in ...`) **并不是**元组推导式，而是创建一个**生成器对象 (Generator Expression)**。
如果需要通过推导逻辑创建元组，必须显式调用 `tuple()`：

```
# 错误理解：以为这是元组
g = (i * 2 for i in range(3))
print(type(g))  # <class 'generator'>

# 正确创建元组
t = tuple(i * 2 for i in range(3))
print(t)        # (0, 2, 4)
```

9.5 进阶：元组真的” 不可变” 吗？

元组的不可变性指的是**“其存储的引用（内存地址）不可变”**。如果元组内部包含可变对象（如列表），该内部对象的内容是可以修改的。

```
t = (1, [2, 3])
# t[0] = 2        # 报错！TypeError（不能修改引用）
# t[1] = [4]      # 报错！TypeError（不能修改引用）

t[1].append(4)    # 合法！修改了内部列表的内容
print(t)          # (1, [2, 3, 4])
```

Listing 10: 元组内含可变对象

10 字典 (Dict) - 键值映射

字典是 Python 中唯一的内置映射类型，存储无序的（Python 3.7+ 插入有序）键值对。

10.1 构造方法

除了字面量 `{'a': 1}` 外，考试中常考以下三种构造方式：

1. **关键字参数 (Keyword Args)**: 键必须是合法的标识符字符串。

```python
d = dict(name='Alice', age=20)
# {'name': 'Alice', 'age': 20}

```

2. **Zip 压缩**: 适合将两个列表组合成字典。

```python
keys = ['a', 'b'];
vals = [1, 2]
d = dict(zip(keys, vals))
# {'a': 1, 'b': 2}

```

3. **元组列表**: 包含 (key, value) 对的可迭代对象。

```python
pairs = [('a', 1), ('b', 2)]
d = dict(pairs)

```

10.2 核心限制：Key 必须 Hashable

[!] 考试指南：TypeError: unhashable type

字典的 **Key** 必须是**不可变类型**（准确说是可哈希的）。

- **合法 Key**: `int`, `float`, `str`, `tuple` (前提是 tuple 内元素也 hashable)。
- **非法 Key**: `list`, `dict`, `set`。

注：*Value* 没有任何限制，可以是任意对象。

10.3 访问、修改与查询

- **访问/新增**: `d[key] = value`。如果 Key 不存在则新增，存在则覆盖。
- **查询**: `key in d`。仅判断 **Key** 是否存在，不检查 Value。
- **视图方法 (View Objects)**:
  - `.keys()`: 返回所有键。
  - `.values()`: 返回所有值。
  - `.items()`: 返回所有 (key, value) 元组。常用于遍历：

```python
for k, v in d.items():
    print(k, v)

```

10.4 字典合并 (Merging)

合并是高频考点，需区分“原地修改”和“返回新对象”。

语法	类型	说明
`a.update(b)`	**原地修改**	把 b 的内容更新到 a 中，返回 None。
`{**a, **b}`	**新对象**	解包合并，如果有重复 Key，b 覆盖 a。
`a | b`	**新对象**	(Python 3.9+) 联合运算符，功能同上。

10.5 删除操作

- `del d[key]`: 删除指定键，Key 不存在报 `KeyError`。
- `.pop(key, [default])`: 删除并**返回**该 Key 对应的值。推荐使用，可提供 default 避免报错。
- `.popitem()`: **LIFO** (后进先出) 删除并返回 `(key, value)` 元组。
- `.clear()`: 清空字典。

10.6 字典推导式 (Dict Comprehension)

语法：`{key_expr: val_expr for item in iterable}`

```python
# 交换键值对
old = {'a': 1, 'b': 2}
new = {v: k for k, v in old.items()}
# {1: 'a', 2: 'b'}
```

Listing 11: 推导式示例

10.7 拷贝机制

与列表一致，**字典的构造函数和 copy 方法均为浅拷贝 (Shallow Copy)**。

```
d1 = {'a': [1, 2]}
d2 = dict(d1)       # 浅拷贝

d2['a'].append(3)   # 修改内部可变对象
print(d1['a'])      # [1, 2, 3] -> d1 受影响
```

11 集合 (Set) - 无序不重复集

集合是**无序**且**不重复**的元素集合，主要用于去重和数学集合运算。

11.1 表示与构造

- **表示**: 使用花括号 `{1, 2, 3}`。
- **构造函数**: `set(iterable)`。

[!] 考试指南：空集合的陷阱

千万小心: `{}` 表示的是**空字典 (Empty Dict)**，而不是空集合！
创建空集合必须使用构造函数：`empty_set = set()`。

11.2 元素限制：必须 Hashable

与字典的 Key 一样，**集合内的元素必须是不可变类型 (Hashable)**。

- **合法**: `{1, (2, 3), "abc"}`
- **非法**: `{ [1, 2] }` (列表不可哈希), `{ {1} }` (集合本身也不可哈希)。

11.3 集合运算 (重点)

集合支持丰富的数学运算符号。

11.4 比较运算

集合的比较是基于**包含关系**的：

- `a <= b`: 判断 a 是否是 b 的**子集**。

符号	含义	说明
`a | b`	**并集** (Union)	包含 a 和 b 中所有的元素。
`a & b`	**交集** (Intersection)	同时存在于 a 和 b 中的元素。
`a ^ b`	**对称差集**	只在 a 或只在 b 中的元素 (异或)。
`a - b`	**差集**	在 a 中但不在 b 中的元素。

- `a < b`: 判断 a 是否是 b 的**真子集**。
- `a >= b`: 判断 a 是否是 b 的**超集**。

11.5 常用方法

- `.add(x)`: 添加元素 x。如果已存在则无视。
- `.remove(x)`: 删除元素 x。如果不存在会报错 `KeyError`。
- `.discard(x)`: 删除元素 x。如果不存在**不会**报错 (安全删除)。

11.6 集合推导式 (Set Comprehension)

语法与字典推导式类似，但只有值没有键。

```
nums = {1, 2, 3}
squares = {x**2 for x in nums}  # {1, 4, 9}
```

12 容器类型综合考点 (List/Tuple/Dict/Set)

以下三条规则是 Python 容器系统的核心机制，考试中常用于考察复杂的嵌套结构。

12.1 1. 嵌套规则 (Nesting Rules)

List, Dict, Set, Tuple 可以任意互相嵌套，但必须遵守 **“底层的 Hashable 限制”**：

1. **List/Tuple**: 内部可以放任意对象（包括 List, Set, Dict）。
   - `[ {1,2}, [3,4] ]` ✓ 合法 (列表里放集合和列表)。
2. **Dict Key / Set Element**: 内部**必须**是不可变对象 (Hashable)。
   - `{ (1,2): "ok" }` ✓ 合法 (元组作 Key)。
   - `{ [1,2]: "no" }` × **报错** (列表作 Key)。
   - `set([ [1], [2] ])` × **报错** (集合里不能放列表)。

12.2 2. 构造函数与迭代行为

List, Set, Tuple 的构造函数均接受**任意 Iterable 对象**。
特别注意 Dict 的迭代行为：当把一个字典传给 `list()` 或 `set()` 时，默认迭代的是 **Keys**。

```
d = {'a': 1, 'b': 2}
lst = list(d)
print(lst)  # ['a', 'b'] (忽略了 Value)
```

Listing 12: 字典转列表

12.3 3. 构造函数总是创建新对象

所有的内置构造函数 (`list()`, `set()`, `dict()`) 在调用时，**一定会**在内存中创建一个全新的对象 (New Identity)。

```
a = [1, 2, 3]
b = list(a)      # 这是一个浅拷贝 (Shallow Copy)

print(a == b)    # True  (内容相等)
print(a is b)    # False (ID不同, 是两个独立容器)
```

13 字符串 (String) - 文本处理

字符串是**不可变 (Immutable)** 的字符序列。这意味着你不能直接修改字符串中的某个字符（如 `s[0] = 'a'` 是非法的）。

13.1 索引与切片

操作方式与列表、元组完全一致。

- **索引**: `s[0]`, `s[-1]`。
- **切片**: `s[start:stop:step]`。
- *注意*: 切片操作会返回一个新的字符串对象。

13.2 常用方法 (Methods)

.find/ .index 查找子串 `sub` 的索引。

- `.find()`: 找不到返回 **-1** (安全)。
- `.index()`: 找不到抛出 **ValueError**。

.count(sub) 统计子串出现的非重叠次数。

.replace 替换子串。
注意: 返回**新字符串**，原字符串不会变!

.split(sep) 将字符串按 `sep` 分割成**列表 (List)**。
`"a,b,c".split(",")` → `['a', 'b', 'c']`

.join(iterable) 将可迭代对象（如列表）拼接成字符串。
语法: `" 分隔符".join(列表)`
`"-".join(['a', 'b'])` → `"a-b"`

.upper/ .lower 全大写 / 全小写转换。

13.3 字符串格式化 (String Formatting)

13.3.1 1. 两种主要语法

1. **.format() 方法**:

```
"Name: {}, Age: {}".format("Alice", 20)
"Name: {0}, Age: {1}".format("Alice", 20) # 指定位置

```

2. **f-string (推荐)**: 在字符串前加 `f`，直接嵌入变量。

```
name = "Alice";
age = 20
f"Name: {name}, Age: {age}"

```

13.3.2 2. 格式说明符 (Format Specifiers)

语法格式：`{value: 格式控制符}`。以下是考试必考的格式代码：

符号	含义	示例 (设 x=10, pi=3.14159)
`:.2f`	保留 2 位小数 (四舍五入)	`f"{pi:.2f}"` → `"3.14"`
`:b`	二进制 (Binary)	`f"{x:b}"` → `"1010"`
`:o`	八进制 (Octal)	`f"{x:o}"` → `"12"`
`:x`	十六进制 (Hex)	`f"{x:x}"` → `"a"`
`:10`	指定宽度 (默认右对齐)	`f"{x:5}"` → `" 10"`
`:<10`	左对齐	`f"{x:<5}"` → `"10 "`
`:>10`	右对齐	`f"{x:>5}"` → `" 10"`
`:^10`	居中对齐	`f"{x:^5}"` → `" 10 "`

13.4 特殊字面量

13.4.1 1. 原始字符串 (Raw String)

在字符串前加 `r` 或 `R`。

作用: **忽略转义字符**（如 `\n`, `\t`）的特殊含义，将其视为普通文本。

常用场景: 正则表达式、Windows 文件路径。

```
print("a\nb")  # 输出两行
print(r"a\nb") # 输出原样: a\nb
```

13.4.2 2. 三引号 (Triple Quotes)

使用 `"""..."""` 或 `'''...'''`。

作用: 允许字符串跨越多行。常用于编写函数或类的文档字符串 (Docstring)。

```
s = """Line 1
Line 2"""
```

14 面向对象编程 (Class)

14.1 基本定义与 Self

Python 中一切皆对象。定义类使用 `class` 关键字。

- **构造函数**: `__init__(self, ...)`。在实例化对象时自动调用，用于初始化属性。
- **self**: 代表**实例对象本身**（类似于 Java/C++ 的 this）。
- **成员函数**: 定义在类内部，第一个参数必须是 `self`。

[!] 考试指南：忘记写 self 的后果

如果在类内部定义函数时忘记加 `self`：

```
class A:
    def func():  # 错误！没有 self
        print("Hi")
a = A()
a.func() # 报错！TypeError
# 原因：a.func() 等价于 A.func(a)，但定义中不接受参数
```

14.2 类变量 vs 实例变量

- **实例变量 (Instance Variable)**: 绑定在 `self` 上（如 `self.x`），每个对象独有。
- **类变量 (Class Variable)**: 定义在类体中（函数之外），所有实例**共享**。

```
class Dog:
    kind = 'Canine'       # 类变量（共享）
    def __init__(self, name):
        self.name = name  # 实例变量（独有）

d1 = Dog('A');
d2 = Dog('B')
d1.kind = 'Wolf'          # 注意！这不会修改类变量，而是创建了一个新的实例变量 d1.kind
print(d1.kind)            # Wolf
print(d2.kind)            # Canine（依然共享类的原值）
print(Dog.kind)           # Canine
```

Listing 13: 类变量的遮蔽 (Shadowing)

14.3 方法类型：@classmethod

- **实例方法**: `def f(self): ...`
- **类方法**: 使用 `@classmethod` 装饰器。第一个参数约定为 `cls` (代表类本身，而非实例)。常用于实现**工厂模式**。

14.4 魔术方法 (Magic Methods)

Python 通过双下划线方法实现运算符重载和特定行为。

14.4.1 1. 字符串表示

- `__str__(self)`: 用户友好的字符串。被 `print()` 和 `str()` 调用。
- `__repr__(self)`: 开发者视角的字符串（通常用于调试）。被交互式命令行直接回显调用。
- 原则: 如果只定义了 `__repr__`，`__str__` 也会默认调用它。

14.4.2 2. 比较运算

- `__eq__` (==), `__ne__` (!=)
- `__lt__` (<), `__le__` (<=), `__gt__` (>), `__ge__` (>=)

14.4.3 3. 算术运算 (重点)

- `__add__(self, other)`: 实现 `self + other`。
- `__radd__(self, other)`: 实现 `other + self` (右加)。
  触发机制: 当左操作数不支持 `__add__` 时,Python 会尝试调用右操作数的 `__radd__`。
- `__iadd__(self, other)`: 实现 `self += other` (就地修改)。

14.4.4 4. 容器模拟

- `__getitem__(self, key)`: 实现 `obj[key]` 读取。
- `__setitem__(self, key, value)`: 实现 `obj[key] = value` 写入。

14.5 继承 (Inheritance)

- 语法: `class Child(Parent): ...`
- **super()**: 用于调用父类的方法，特别是构造函数。

```python
class Parent:
    def __init__(self):
        self.x = 1

class Child(Parent):
    def __init__(self):
        super().__init__() # 必须显式调用，否则父类属性不会初始化
        self.y = 2
```

15 模块 (Module)

15.1 导入语法 (Import Syntax)

- `import math`: 使用 `math.sqrt(4)` 调用。
- `import math as m`: 别名，使用 `m.sqrt(4)`。
- `from math import sqrt`: 直接导入符号，使用 `sqrt(4)`。
- `from math import *`: 导入所有（不推荐，易污染命名空间）。

15.2 导入机制 (Execution & Caching)

Import 发生了什么?

1. **执行代码**: `import A` 会将模块 A 中的顶层代码**从头到尾执行一遍**。
2. **缓存机制**: Python 会把导入过的模块缓存在 `sys.modules` 中。
   多次 import 只执行一次: 如果在一个程序中多次写了 `import A`，后续的导入直接使用缓存，**不会**重新执行模块代码（除非手动使用 `importlib.reload`）。

15.3 `__name__` 与 `'__main__'`

每个模块都有一个内置属性 `__name__`。

- 如果模块是被 **直接运行**的 (例如 `python my_script.py`):
  `__name__` 的值为 `'__main__'`。
- 如果模块是被 **导入**的 (例如 `import my_script`):
  `__name__` 的值为模块本身的名字 (即 `'my_script'`)。

经典用法 (测试代码保护):

```python
def func():
    print("Function logic")

# 以下代码只有在直接运行此文件时才执行
# 被别人 import 时不会执行
if __name__ == '__main__':
    func()
```

16 异常处理与断言 (Exception & Assertion)

16.1 异常捕获机制

Python 使用 `try...except...else...finally` 结构来处理运行时错误。

```python
try:
    # 可能抛出异常的代码
    res = 1 / 0
except ZeroDivisionError:
    # 当捕获到特定异常时执行
    print("除数不能为0")
except (IndexError, KeyError) as e:
    # 捕获多种异常，并获取异常对象 e
    print(f"索引或Key错误: {e}")
else:
    # 【仅在】try 块没有抛出任何异常时执行
    print("一切正常")
finally:
    # 【无论如何】都会执行 (常用于关闭文件、释放锁)
    print("清理工作")
```

Listing 14: 完整的异常结构

16.2 finally 的绝对执行权 (高频考点)

finally 块具有极高的优先级。即使在 try 块中执行了 return, break 或 continue，finally 依然会在跳转前被执行。

[!] 考试指南：循环中的 finally 与 break

考题常考：break 是否会跳过 finally？答案是**不会**。

```python
def test_loop():
    for i in range(3):
        try:
            if i == 1:
                print("即将 Break")
                break  # 准备跳出
        finally:
            print(f"Finally 执行: {i}")

test_loop()
# 输出顺序:
# Finally 执行: 0
# 即将 Break
# Finally 执行: 1  <-- 即使 break 了, finally 依然执行!
```

16.3 常见异常类型速查

考试中常给出一段报错代码，要求判断抛出什么类型的异常。

异常类型	触发场景与示例
ZeroDivisionError	除数为 0。 1 / 0, 1 % 0
IndexError	序列索引越界 (List/Tuple/String)。 lst = [1]; lst[10]
KeyError	字典中查找不存在的键。 d = {'a':1}; d['b']
ValueError	**类型正确但数值不合法。** int("abc") (字符串不能转数字) list.remove(x) (删除不存在的元素)
TypeError	**操作类型不匹配。**

	`1 + "1"` (整数加字符串) `len(5)` (整数没有长度) `list((1))` (整数不可迭代，构造失败)
`AttributeError`	访问对象不存在的属性或方法。 `[].add(1)` (列表没有 add 方法，只有 append)
`NameError`	使用了未定义的变量名。 `print(undefined_var)`

`UnboundLocalError`	**作用域陷阱：**在函数内引用局部变量前对其赋值（导致全局屏蔽）。
	`x = 10`
	`def f():`
	`  print(x) # 报错！Python发现下面有赋值，`
	`  x = 5  # 认为x是局部变量，但print时还没赋值`

`AssertionError`	`assert` 语句失败时抛出。

16.4 断言 (Assertion)

用于调试和防御性编程，确保程序在某个特定状态下是正确的。如果条件为假，程序崩溃。

- **语法**: `assert expression [, message]`
- **逻辑**: 等价于：

```
if not expression:
    raise AssertionError(message)
```

```
1 def apply_discount(price, discount):
2     # 确保价格和折扣合理，否则直接报错
3     assert price >= 0, "Price cannot be negative"
4     assert 0 <= discount <= 1, "Discount must be 0-1"
5     return price * (1 - discount)
```

Listing 15: Assert 使用示例

17 Random 随机数模块

使用前需导入：`import random`。

17.1 数值生成 (Floats & Integers)

.random() 返回 `[0.0, 1.0)` 之间的浮点数 (包含 0，不包含 1)。

.uniform(a, b) 返回 `[a, b]` 之间的浮点数 (通常包含边界 b)。

.randint(a, b) 返回 `[a, b]` 之间的整数。
注意: **包含**右边界 `b`！这是 Python 中少有的闭区间。

.randrange 从 `range(start, stop, step)` 中随机选取一个整数。
注意: **不包含**右边界 `stop` (左闭右开)。

[!] 考试指南：边界陷阱：randint vs randrange

- `randint(1, 3)` 可能返回: 1, 2, **3**。
- `randrange(1, 3)` 可能返回: 1, 2 (绝不会返回 3)。

17.2 序列操作 (Sequences)

假设序列为 `seq = [1, 2, 3, 4, 5]`。

.choice(seq) 从序列中随机返回**一个**元素。

.shuffle(seq) **就地打乱 (In-place)** 序列顺序。
返回 None！ 千万不要写 `seq = random.shuffle(seq)`。

.sample(seq, k) 从序列中随机抽取 k 个元素。
特征: **无放回抽样** (No Replacement)。结果中元素不重复。
限制: k 不能大于序列长度 `len(seq)`，否则报错。

.choices(seq, k) (Python 3.6+) 从序列中随机抽取 k 个元素。
特征: **有放回抽样** (With Replacement)。结果可能重复。
功能: 支持权重参数 `weights`。

sample vs choices

- 想要“不重复”的抽奖？用 **sample**。
- 想要“抛硬币/掷骰子”（独立重复实验）？用 **choices**。

17.3 随机种子 (Reproducibility)

.seed(a=None) 初始化伪随机数生成器。
意义: 如果种子相同，生成的随机数序列**完全一致**。常用于调试或作业中固定结果。

```
random.seed(10)
print(random.random())  # 假设输出 0.571...

random.seed(10)         # 再次设置相同的种子
print(random.random())  # 输出完全一样 0.571...
```

Listing 16: Seed 的效果

18 文件操作 (File I/O)

18.1 字节序列 (Byte Sequence)

在计算机底层，所有文件本质上都是字节。Python 提供了专门处理二进制数据的类型。

- **bytes (不可变)**:
  语法: `b'hello'` 或 `bytes([65, 66, 67])`。
  只能包含 ASCII 字符或十六进制转义符 (如 `\xff`)。
- **意义**: 用于处理二进制文件（图片、音频）或网络数据包。
- **转换**:
  `str.encode('utf-8')` → `bytes`
  `bytes.decode('utf-8')` → `str`

18.2 打开与关闭 (Open & Close)

18.2.1 1. open() 函数

语法: `f = open(file, mode='r', encoding='utf-8')`

[!] 考试指南：'w' 模式的危险性

使用 `'w'` 模式打开文件时，**无论是否写入内容，原文件的内容都会立即被清空。**
如果只想修改部分内容或追加，请使用 `'r+'` 或 `'a'`。

模式	说明
'r'	只读 (默认)。如果文件不存在抛出 FileNotFoundError。
'w'	只写。如果文件存在，先清空内容 (Truncate) 再写入；不存在则创建。
'a'	追加。写入的数据会被加到文件末尾。
'b'	二进制模式 (Binary)。如 'rb', 'wb'。读写对象为 bytes 而非 str。
'+'	更新模式 (读写)。如 'r+' (读写，不清除), 'w+' (读写，先清除)。

18.2.2 2. with 语句 (Context Manager)

即使发生异常，也能保证文件被正确关闭。这是考试和实战的**标准写法**。

```
# 不推荐
f = open("data.txt", "r")
data = f.read()
f.close() # 如果上面出错，这行可能不执行 -> 资源泄露

# 推荐 (with)
with open("data.txt", "r") as f:
    data = f.read()
# 离开缩进块后，f.close() 会自动被调用
```

Listing 17: 自动关闭机制

18.3 读写方法

假设文件对象为 f。

18.3.1 1. 读取 (Read)

- f.read(size=-1): 读取整个文件（或指定 size 个字符/字节）。返回字符串。
- f.readline(): 读取**一行**（包含末尾的 \n）。读到 EOF 返回空字符串 ''。
- f.readlines(): 读取所有行，返回一个**列表** ['Line1\n', 'Line2\n']。

最佳遍历方式

不要用 readlines() 遍历大文件（耗内存）。直接在文件对象上迭代：

```
for line in f:
    print(line.strip()) # 高效，逐行读取
```

18.3.2 2. 写入 (Write)

- f.write(s): 将字符串 s 写入文件。返回写入的字符数。
- f.writelines(lines): 将字符串列表写入文件。

[!] 考试指南：writelines 不会自动换行

writelines 不会在每个元素后面自动添加换行符，需要手动处理。

```
lines = ["A", "B"]
f.writelines(lines) # 写入 "AB"
f.writelines([l + '\n' for l in lines]) # 写入 "A\nB\n"
```

18.4 指针操作与缓冲

- f.tell(): 返回当前文件指针的位置 (字节偏移量)。
- f.seek(offset, whence=0): 移动指针。
  - whence=0: 从文件开头算 (默认)。
  - whence=1: 从当前位置算 (仅二进制模式)。
  - whence=2: 从文件末尾算 (仅二进制模式)。
  - 常用: f.seek(0) (回到开头)。
- f.flush(): 强制将缓冲区的数据写入硬盘，不关闭文件。

19 迭代器与解包 (Iterable & Iterator)

19.1 迭代器协议

Python 的 for 循环底层依赖于迭代器协议。

- **可迭代对象 (Iterable)**:
  实现了 __iter__() 方法的对象。例如 List, Tuple, Dict, Str。
  作用: 可以被 for 循环遍历，或者通过 iter(obj) 获取迭代器。

- **迭代器 (Iterator)**:
  实现了 `__iter__()` 和 `__next__()` 方法的对象。
  作用: 负责维护遍历的状态（游标）。

19.2 iter() 与 next() 的工作机制

1. `it = iter(iterable)`: 调用对象的 `__iter__()`，返回一个迭代器。
2. `val = next(it)`: 调用迭代器的 `__next__()`，返回下一个值。
3. **StopIteration**: 当没有更多元素时，`next()` 会抛出此异常，通知 `for` 循环停止。

```
s = "ABC"
it = iter(s)          # 获取迭代器
while True:
    try:
        val = next(it)  # 获取值
        print(val)
    except StopIteration:
        break           # 捕获异常退出循环
```

Listing 18: 模拟 for 循环底层实现

19.3 解包 (Unpacking)

- **序列解包 (*a)**: 用于 List/Tuple。

```
head, *mid, tail = [1, 2, 3, 4, 5]
# head=1, mid=[2,3,4], tail=5

```

- **字典解包 (**b)**: 用于函数参数或字典合并。

```
def func(a, b): pass
d = {'a': 1, 'b': 2}
func(**d)  # 等价于 func(a=1, b=2)

```

20 结构化模式匹配 (Match Statement)

Python 3.10+ 引入的新特性，类似于 Switch-Case 但更强大，支持结构解构。

```python
point = (0, 10)

match point:
    case (0, 0):
        print("Origin")
    case (0, y):  # 捕获 y
        print(f"Y-axis at {y}")
    case (x, 0):  # 捕获 x
        print(f"X-axis at {x}")
    case (x, y) if x == y: # 带守卫条件 (Guard)
        print("Diagonal")
    case _:
        print("Something else")
```

Listing 19: Match 语法示例

21 生成器 (Generator)

生成器是一种特殊的迭代器，通过函数动态生成值，**省内存**。

21.1 基本语法：yield

如果函数中包含 `yield` 关键字，该函数就变成了**生成器函数**。

- **return**: 终止函数，返回结果，销毁局部变量。
- **yield**: **暂停**函数执行，返回结果，**保留**局部变量状态。下次调用 `next()` 时从暂停处继续执行。

```python
def count_down(n):
    while n > 0:
        yield n   # 产出 n 并暂停
        n -= 1

g = count_down(3)
print(next(g)) # 3
print(next(g)) # 2
```

21.2 生成器表达式 (Generator Expression)

[!] 考试指南：不是元组推导式

使用圆括号 () 包裹的推导式是生成器表达式。Python 中**不存在** Tuple Comprehension。

```
g = (x**2 for x in range(10))
# <generator object ...>
```

22 函数高级特性

22.1 变量作用域 (Scope)

遵循 **LEGB** 规则：Local → Enclosing (闭包) → Global → Built-in。

22.1.1 修改外部变量

默认情况下，在函数内赋值会创建新的局部变量。如需修改外部变量，需声明：

- **global x**: 声明 x 是**全局变量**。
- **nonlocal x**: 声明 x 是**外层嵌套函数**（非全局）的变量。

```
count = 0  # Global

def outer():
    count = 10  # Enclosing (Outer Local)

    def inner():
        nonlocal count
        count += 1  # 修改的是 outer 的 count (变为 11)

        # global count # 语法错误! 不能同时声明
        # 若用 global count, 则修改的是最上面的 count (变为 1)

    inner()
    print(count) # 11
```

Listing 20: global vs nonlocal

22.2 嵌套定义与闭包

函数内部可以定义函数，并返回内部函数。如果内部函数引用了外部函数的变量，则形成**闭包 (Closure)**。

22.3 装饰器 (Decorator)

装饰器本质上是一个**高阶函数**：接收一个函数，返回一个新函数（通常是对原函数的增强）。

- **语法糖**: `@decorator`

```python
def my_decorator(func):
    def wrapper():
        print("Before calling")
        func()
        print("After calling")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

# 等价于:
# say_hello = my_decorator(say_hello)
```

Listing 21: Decorator 原理