Python基础内容快查

一、python容器

🔴 Python 核心容器

类型	是否有序	是否可变	典型用途
list	✅	✅	动态数组
tuple	✅	❌	只读序列 / hash key
str	✅	❌	字符序列
dict	✅	✅	键值映射
set	❌	✅	去重 / 集合运算
frozenset	❌	❌	不可变集合

🟠 统一特性

所有序列都支持的操作

```
len(x)
x[index]
x[start:end:step]
for elememt in container
in / not in
```

函数/操作	操作
数值 / 长度	
len(x)	尾插
min(x)/max(x)	批量追加
sum(x)	指定位置插
判断	
any(iterable)	尾插
all(iterable)	批量追加
排序 / 枚举	
sorted(iterable)	尾插
reversed(iterable)	批量追加
enumerate(iterable)	指定位置插
映射 / 过滤（函数式）	

函数/操作	操作
map(func, iterable)	尾插
filter(func, iterable)	批量追加
zip(a, b)	指定位置插

🟡 各类容器常用操作

1、序列容器

1️⃣ 列表(list)

函数/操作	操作
a.append(x)	尾插
a.extend(iter)	批量追加
a.insert(i, x)	指定位置插
a.pop()	删除末尾
a.pop(i)	删除指定索引
a.remove(x)	删除第一个 x
del a[i:j]	删除区间
a.clear()	清空
a.index(x)	查询下标
a.count(x)	查询x个数

2️⃣ 元组(tuple)

解包常用!!

```
a, b, c = (1, 2, 3)
x, *mid, y = (1, 2, 3, 4, 5)
```

3️⃣ 字符串(string)

函数/操作	操作
s.startswith("he")	判断是否以什么开头
s.endswith("lo")	判断是否以什么结尾
s.isdigit()...	判断类型
s.find()...	查找内容，找不到返回 -1
s.index("x")	查找内容，找不到抛异常
s.replace("a", "b")	替换

函数/操作	操作
s.split(",")	按xxx分割，返回list
s.(l/r)strip()	去空白

2、映射容器（Mapping）

4️⃣ 字典(dict)

函数/操作	操作
d.update({"d": 4})	修改
d.get("x")	查询，不存在 → None
d.get("x", 0)	查询，指定默认值
d["x"]	⚠️ KeyError
d.clear()	清空字典
d.pop("a"))	删除key
d.popitem()	删除最后插入的

3、集合容器（Set）

5️⃣ 集合(set) & 不可变集合(frozenset)

函数/操作	操作
s.add(x)	增
s.remove(x)	删，不存在 → KeyError
s.discard(x)	删，不存在 → 安全
s.pop()	删
a.union(b)	求并集
a.intersection(b)	求交集

- 不可变集合有部分操作不可用

注：

1. a | b # 并集
2. a & b # 交集
3. a - b # 差集
4. a ^ b # 对称差

二、函数

🔴 Lambda

- 基本语法：**lambda 参数1, 参数2, ... : 表达式**

⚠️ **注意两点：**

1. 冒号 : 右边必须是一个表达式（expression） 不能写多条语句（statement），比如 print + return、if ...: ... 这种语句式结构都不行。
2. lambda 自动返回表达式的值，不写 return。

1️⃣ 闭包与晚绑定

> **Python 的 lambda / 内部函数，捕获的是“变量”，不是“当时的值” 而且这个变量是在函数真正被调用时才取值 —— 这就叫晚绑定（late binding）。**

⚠️ **典型例子**

```
funcs = []
for i in range(3):
    funcs.append(lambda: i) # 此时funcs的每一项lambda都存入了i的引用，而非值

print([f() for f in funcs]) # f的类型为函数对象，f()就是调用函数的意思
```

此时的**返回值为:[2,2,2]**

✅ 修复方式：用默认参数“冻结”值

```
funcs = []
for i in range(3):
    funcs.append(lambda i=i: i) # 定义即求值，等价def f(i=i): return i

print([f() for f in funcs]) # funcs = [<function>, <function>, <function>]
```

此时的**返回值为:[0,1,2]**

三、类

🔴 魔法方法（dunder methods）

> 此处主要引用Python开发必备：解锁魔法方法的实用技巧大全，文章作者：AInLife，可用于查看详细例子。

1、对象生命周期管理

1️⃣ **new**(cls, *args, **kwargs)

- 控制对象的创建，是真正的**构造器**，是对象创建过程中的第一个方法（静态方法），它在__init__之前调用。**通常情况下，是不需要直接使用__new__**，但如果你需要控制对象的创建过程（如：实现单例模式或自定义元类），可以重写它。

2️⃣ **init**(self, *args, **kwargs)

- 是**初始化器**，初始化对象的，对象在内存已经分配，即self已经存在，接收参数并设置初始状态，它在每次**创建对象时被自动调用**，几乎每个类都会实现它。

3️⃣ **del**(self)

- 定义对象被垃圾回收时的行为。需要注意的是，最好谨慎使用，如果使用不当，可能引发资源管理问题。因为**该方法不保证一定被调用**，其主要依赖垃圾回收机制的策略。

2、对象表示与格式化

1️⃣ **str**(self)

- **为最终用户设计**，要求可读性好，这个方法应该返回一个易于理解的字符串，用于**展示对象的“外观”**。

2️⃣ **repr**(self)

- **为开发者设计**，要求**明确无误**，定义repr(obj)或交互环境中对象的表示，通常用于开发者调试，返回详细信息，它的目标是生成一个可以通过eval()恢复的字符串（即反向构造对象）。

3️⃣ **format**(self, format_spec)

3、属性访问控制

1️⃣ **getattr**(self, name)

- 当**访问不存在的属性**时调用

2️⃣ **setattr**(self, name, value)

- **设置属性**时调用，常用于**拦截赋值**

3️⃣ **delattr**(self, name)

- **删除属性**时调用

4️⃣ **getattribute**(self, name)

- **无条件访问任何属性**时都会调用

4、容器与迭代器

1️⃣ **len**(self)

- 当**访问不存在的属性**时调用

2️⃣ **getitem**(self, key)

- 可以实现自定义对象的**下标访问行为**

3️⃣ **setitem**(self, key, value)

- **删除属性**时调用

4️⃣ **delitem**(self, key)

- **无条件访问任何属性**时都会调用

5️⃣ **iter**(self)、**next**(self)

- **无条件访问任何属性**时都会调用

6️⃣ **contains**(self, item)

- **无条件访问任何属性**时都会调用

5、同步-上下文管理

1️⃣ **enter**(self)

- **进入with时调用**，返回上下文对象。

2️⃣ **exit**(self, exc_type, exc_value, traceback)

- **退出with时调用**，处理异常，需要提醒的是，该方法返回True时可抑制异常，也就是说异常会被“吞掉”，上下文管理器外的代码不会收到异常。这是一个强大但危险的特性。很适合事务处理，如果结合contextlib模块可简化上下文管理器的实现。

6、异步-上下文管理

1 **await**(self)

- (可等待对象)必须返回一个迭代器（通常通过 iter() 包装一个生成器）,可用于**实现自定义可等待对象**

2 **alter**(self)

- (异步迭代器)返回异步迭代器对象，通常return self。

3 **anext**

- 必须返回一个awaitable 对象，（通常是 coroutine 或 Task）

4 **aenter**(self)

- (异步上下文管理器)：返回进入时的资源，常为await self.connect().

5 **aexit**(self, exc_type, exc_value, traceback)

- 用于异步清理（如关闭连接），可处理异常

7、可调用对象

1 **call**(self, *args, **kwargs)

- **让类的实例像函数一样被调用**，适合实现装饰器或函数式接口。

8、描述符协议

1 **get**(self, instance, owner)

- **获取属性值。**

2 **set**(self, instance, value)

- **设置属性值。**

3 **delete**(self, instance)

- **输出属性。**

4️⃣ **set_name**(self, owner, name)

- 在类创建时**设置描述符名称**（Python3.6+）

9、运算符重载

1️⃣ 实现算术运算符

- `__add__(self, other)`：实现加法运算符 `+`
- `__sub__(self, other)`：实现减法运算符 `-`
- `__mul__(self, other)`：实现乘法运算符 `*`
- `__truediv__(self, other)`：实现真除法运算符 `/`
- `__floordiv__(self, other)`：实现整数除法运算符 `//`
- `__mod__(self, other)`：实现取模运算符 `%`
- `__pow__(self, other[, modulo])`：实现幂运算符 `**`
- `__iadd__(self, other)`：实现复合运算符 `+=`
- `__isub__(self, other)`：实现复合运算符 `-=`
- `__imul__(self, other)`：实现复合运算符 `*=`

2️⃣ 实现位运算符（不常用）

- `__lshift__(self, other)`：实现左移位运算符 `<<`
- `__rshift__(self, other)`：实现右移位运算符 `>>`
- `__and__(self, other)`：实现按位与运算符 `&`
- `__xor__(self, other)`：实现按位异或运算符 `^`
- `__or__(self, other)`：实现按位或运算符 `|`

3️⃣ 实现比较运算符

- `__lt__(self, other)`：实现小于运算符 `<`
- `__le__(self, other)`：实现小于等于运算符 `<=`
- `__eq__(self, other)`：实现等于运算符 `==`
- `__ne__(self, other)`：实现不等于运算符 `!=`
- `__gt__(self, other)`：实现大于运算符 `>`

- `__ge__(self, other)`：实现大于等于运算符 `>=`

友情提醒：
在实现比较运算符时，推荐使用 `functools.total_ordering` 装饰器，只需定义 `__eq__` 和 `__lt__`，即可自动生成其余比较方法。

4️⃣ 实现容器类型相关方法（序列 / 映射）

- `__getitem__(self, key)`：通过索引或键获取元素，如 `obj[key]`
- `__setitem__(self, key, value)`：通过索引或键设置元素，如 `obj[key] = value`
- `__delitem__(self, key)`：通过索引或键删除元素，如 `del obj[key]`
- `__contains__(self, item)`：实现成员运算符 `in`
- `__len__(self)`：实现内置函数 `len()`，返回容器长度

10、一些不常见但很有用的方法

1️⃣ slots

- **限制类可添加的属性**，优化内存和性能。

2️⃣ class

- **获取或修改对象的类**，动态性极强。

3️⃣ copy(self)

- **浅拷贝**

4️⃣ deepcopy(self, memo)

- **深拷贝**

5️⃣ getstate(self)

- pickle 序列化时获取对象状态

6️⃣ setstate(self, state)

- pickle 反序列化时恢复状态

7️⃣ dir(self)

- 自定义属性列表

8️⃣ **sizeof**(self)

- **返回对象内存大小**

9️⃣ **hash**(self)

- 用于哈希表（如字典键）

🔟 **instancecheck**(self, instance)、**subclasscheck**(self, subclass)

- 自定义 isinstance 和 issubclass 行为，通常在元类中使用。

🟠 装饰器

> 此处询问了GPT并参考了
> Python 常用装饰器全解析，文章作者：加菲大杂烩 和 Python 函数装饰器，文章来源：菜鸟编程 ，可用于查看部分详细例子。

1️⃣ @staticmethod

含义

- 不自动传 `self`
- 不自动传 `cls`
- 就是挂在类里的普通函数,**方法逻辑不依赖对象状态**

2️⃣ @classmethod

含义

- 第一个参数是 `cls`
- 操作的是类变量
- 常用于**工厂方法 / 影响整个类**: 从类的角度做事”，而不是某个对象

3️⃣ @property

含义

- 用 `obj.x` 访问
- 实际调用的是函数
- 用来封装 getter
- 让外界**像访问属性一样**，但内部还能控制逻辑

4️⃣ @xxx.setter

含义

- 搭配 `@property`
- 控制赋值逻辑

5️⃣ @dataclass

含义

- 自动生成 `__init__`
- 自动生成 `__repr__`，`__eq__`
- 非常适合“数据结构类”
- 类主要是“装数据”，不是“重逻辑”

6️⃣ @my_decorator

含义

- 不改函数代码
- 给它**增加行为（日志、权限、计时等）**

7️⃣ @overload

含义

- 用于静态类型提示
- 给函数提供多个签名
- 需配合类型检查器使用。

8️⃣ @final

含义

- 防继承/防重写装饰器

1️⃣ 2️⃣ 3️⃣ 4️⃣ 5️⃣ 6️⃣ 7️⃣ 8️⃣ 9️⃣ 🔟 🔴 🟠 🟡 🟢 🔵 🟣 🟤 ⚫ ⚪ ✅ ❌ ⚠️ ❗ ‼️

emoji_website