Python 基础语法笔记

柯羽昕

2025 年 7 月 28 日

1 基本数据结构

1.1 列表

Python 列表是一种可变序列，可以存放任何类型的元素，且不要求元素维度相同。列表支持多种操作：

```
a = 1
b = "math"
c = ["ml", a, b, ["list"]]
print(c*2)  # 复制一遍列表
c.append("111")  # 在列表末尾添加元素
c.remove("ml")  # 删除列表中的元素
```

输出：

```
['ml', 1, 'math', ['list'], 'ml', 1, 'math', ['list']]
['ml', 1, 'math', ['list'], '111']
```

列表是可变的数据结构，创建后可以修改其内容。`append()` 方法用于在列表末尾添加新元素，`remove()` 方法用于删除指定元素。

1.2 元组

元组与列表相似，但是一旦创建就不可修改。元组适用于存储不应被修改的数据集合：

```
d = ("ml", a, b, ["list"])
```

输出：

```
('ml', 1, 'math', ['list'])
```

虽然元组本身不可变，但如果元组中包含可变元素 (如列表)，这些元素的内容仍然可以改变。

1.3 字典

字典是 Python 中的键值对映射结构，可以理解为带名字的列表：

```
Dict = {"a":1, "b":"math", "c":["ml",1,"math",["list"]]}
Dict["c"] = "new"  # 修改字典值
```

输出：

```
{'a': 1, 'b': 'math', 'c': 'new'}
```

字典通过键来访问值，键必须是不可变类型，而值可以是任意类型。字典的值可以通过赋值语句直接修改。

2 控制结构

2.1 For 循环

Python 的 for 循环可以配合 range() 函数使用，range() 生成一个整数序列：

```
# 打印九九乘法表
for i in range(1,10,1):
    lst = []
    for j in range(1,10,1):
        if i >= j:
            lst.append(f"{i}*{j}={i*j}")
    print(lst)
```

输出：

```
['1*1=1']
['2*1=2', '2*2=4']
['3*1=3', '3*2=6', '3*3=9']
['4*1=4', '4*2=8', '4*3=12', '4*4=16']
['5*1=5', '5*2=10', '5*3=15', '5*4=20', '5*5=25']
['6*1=6', '6*2=12', '6*3=18', '6*4=24', '6*5=30', '6*6=36']
```

```
['7*1=7', '7*2=14', '7*3=21', '7*4=28', '7*5=35', '7*6=42', '7*7=49']
['8*1=8', '8*2=16', '8*3=24', '8*4=32', '8*5=40', '8*6=48', '8*7=56', '8*8=64']
['9*1=9', '9*2=18', '9*3=27', '9*4=36', '9*5=45', '9*6=54', '9*7=63', '9*8=72', '9*9=81']
```

for 循环也可以直接遍历各种数据结构：

```
# 遍历列表
for i in c:
    print(i)

# 遍历字典
for i in Dict.keys():
    print(i)  # 遍历键
for i in Dict.values():
    print(i)  # 遍历值
for i in Dict.items():
    print(i)  # 遍历键值对
```

输出：

```
ml
1
math
['list']
new
a
b
c
math
new
('a', 1)
('b', 'math')
('c', 'new')
```

2.2 条件判断

Python 使用 if 语句进行条件判断，isinstance() 函数可以检查变量类型：

```
if isinstance(a, int):
    print("a是整数")
```

输出：

```
a是整数
```

3 函数与方法

Python 函数使用 def 关键字定义，可以设置默认参数：

```
def method1(a, b=1):  # b是默认参数，必须放在参数列表最后
    if isinstance(a, int):
        print("a是整数")

method1(1)  # 调用方法
```

输出：

```
a是整数
```

注意默认参数必须放在参数列表的最后位置。函数内部只能访问局部变量或全局变量。

4 类与面向对象

4.1 类定义

Python 类使用 class 关键字定义，包含类属性和实例方法：

```
class MyClass:
    i = 123456  # 类属性

    def __init__(self, a, b):  # 初始化方法
        self.a = a  # 实例变量
        self.b = b  # 实例变量

    def method(self):
        return f"Class with a={self.a} and b={self.b}"
```

4.2 类的使用

类实例化后可以访问类属性和实例方法：

```
x = MyClass(1, "math")
print("类属性i为:", x.i)  # 访问类属性
print("方法输出:", x.method())  # 调用实例方法

x.i = 111111  # 修改实例属性
print("修改后类属性:", MyClass.i)  # 类属性不变
print("修改后实例属性:", x.i)  # 实例属性改变
```

输出：

```
类属性i为: 123456
方法输出: Class with a=1 and b=math
修改后类属性: 123456
修改后实例属性: 111111
```

注意修改实例属性不会影响类属性，它们属于不同的命名空间。

4.3 继承

Python 支持类的继承和方法重写：

```
class MySubClass(MyClass):
    def __init__(self, a, b, c):
        super().__init__(a, b)  # 调用父类初始化
        self.c = c  # 新增属性

    def method(self):  # 方法重写
        return f"Subclass with a={self.a}, b={self.b}, and c={self.c}"
```

子类通过 super() 函数调用父类方法，可以添加新属性和重写父类方法。

5 应用：KMeans

5.1 KMeans 实现步骤

KMeans 算法的基本步骤如下：

1. **初始化聚类中心：**从数据中随机选择 k 个样本作为初始聚类中心。
2. **分配每个样本到最近的聚类中心：**对于每个样本，计算它到所有 k 个聚类中心的距离，并把它分配给最近的聚类中心。

3. **更新聚类中心：**对每个簇，重新计算其中心（即该簇中所有点的均值）。

4. **判断是否收敛：**如果聚类中心的变化小于预设阈值，或者达到最大迭代次数，则停止；否则，返回第 2 步继续迭代。

5.2 使用 sklearn 实现 KMeans 聚类

使用 sklearn 库实现 KMeans 聚类。以下代码生成了一些模拟数据并使用 KMeans 进行聚类：

```
import matplotlib as mpl#调用包
import matplotlib.pyplot as plt
import numpy as np
import sklearn.datasets as ds
from sklearn.cluster import KMeans
from sklearn.metrics import homogeneity_score, completeness_score, v_measure_score, adjus
    adjusted_rand_score, silhouette_score

x, y = ds.make_blobs(400, n_features=2, centers=4, random_state=2025)
model = KMeans(n_clusters=4)#可以用Control点击包查看包装的代码
model.fit(x)
y_pred = model.predict(x)
print('y = ', y[:30])
print('y_Pred = ', y_pred[:30])
print('homogeneity_score = ', homogeneity_score(y, y_pred))
print('completeness_score = ', completeness_score(y, y_pred))
print('v_measure_score = ', v_measure_score(y, y_pred))
print('adjusted_mutual_info_score = ', adjusted_mutual_info_score(y, y_pred))
print('adjusted_rand_score = ', adjusted_rand_score(y, y_pred))
print('silhouette_score = ', silhouette_score(x, y_pred))

plt.figure(figsize=(8, 4))
plt.subplot(121)
plt.plot(x[:, 0], x[:, 1], 'r.', ms=3)
plt.subplot(122)
plt.scatter(x[:, 0], x[:, 1], c=y_pred, marker='.', cmap=mpl.colors.ListedColormap(list('
plt.tight_layout()
plt.show()
```

5.3 自定义 KMeans 实现

为了更好地理解 KMeans 算法，我们可以尝试自己实现一个 KMeans 类。以下是自定义 KMeans 的实现：

```
class Own_KMeans:
    def __init__(self, k=3, max_iters=100, tol=1e-4, random_state=None):
        self.k = k
        self.max_iters = max_iters
        self.tol = tol
        self.random_state = random_state
        self.centroids = None
        self.labels = None

    def fit(self, X):
        if self.random_state is not None:
            np.random.seed(self.random_state)

        n_samples, n_features = X.shape

        # Step 1: 初始化聚类中心（随机选k个样本）
        random_idxs = np.random.choice(n_samples, self.k, replace=False)
        self.centroids = X[random_idxs]

        for i in range(self.max_iters):
            # Step 2: 计算所有样本到每个聚类中心的距离，并分配标签
            distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
            self.labels = np.argmin(distances, axis=1)

            # Step 3: 更新聚类中心
            new_centroids = np.array([X[self.labels == j].mean(axis=0) for j in range(sel

            # Step 4: 判断是否收敛
            if np.all(np.abs(new_centroids - self.centroids) < self.tol):
                break
            self.centroids = new_centroids
```

代码解释：1. `fit` 方法实现了 KMeans 的核心步骤，包括初始化聚类中心、计算距离并分配标签、更新聚类中心以及判断收敛。2. 使用 `np.linalg.norm` 计算样本点与聚类中心之间的欧几里得距离，并根据距离

最小的原则分配标签。3. 通过计算标签相同样本的均值来更新聚类中心。4. 判断聚类中心的变化是否小于预设阈值，如果小于则停止迭代，否则继续。