Python程序设计与实践

第八课：图像、文本、嵌套、计时

2025. 4

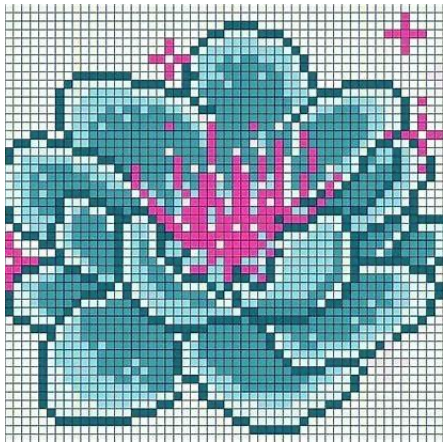
Today

- 图像与动画的处理
- 深入的文本数据的处理
- 嵌套数据类型的处理
- 计时与计数的使用

- 如何使用Python进行基础图像处理和简单动画制作
  - 理解图像的基本组成与像素的原理
  - 掌握使用Python操作和修改静态图像的方法
  - 学会使用tkinter模块进行基本图形绘制与动画制作
  - 理解动画的时间循环机制与坐标变换的实现方式
  - 为后续计算机视觉或游戏开发打下基础

■ 什么是图像?

- 图像由一个个小方格组成，这些小方格称为像素（Pixel）
- 每个像素包含3个颜色分量：红（R）、绿（G）、蓝（B），每个分量范围为0~255
- 图像可看作一个二维矩阵，坐标系原点在左上角，x轴向右延展，y轴向下延展
- 一个像素RGB值为（255, 0, 0）表示纯红色，(0, 255, 0) 表示纯绿色
- 像素操作是图像处理的基础，比如变暗、增强某种颜色、提取特定区域等



图像坐标系统

- 坐标原点(0, 0)位于图像左上角
- x 轴向右增长，y 轴向下增长
- 每个像素在二维坐标中对应唯一(x, y)位置
- 图像在程序中是一个像素矩阵，可通过坐标(x, y)访问

Python中的图像处理工具

- Python常用图像处理库：Pillow
  - Image 是 Pillow 中的核心类，用于打开和操作图像
  - show() 方法会打开默认图像查看器预览图像

```
from PIL import Image

img = Image.open('cat.jpg')

img.show()
```

Python中的图像处理——变暗

- 使用 `Pillow` 的 `ImageEnhance` 模块可以调整图像亮度
- 变暗操作的原理是将亮度乘以一个 `< 1` 的因子
- 示例代码:

```
from PIL import Image, ImageEnhance

img = Image.open('cat.jpg')

enhancer = ImageEnhance.Brightness(img)

darker_img = enhancer.enhance(0.5)   # 亮度降低为原来的一半

darker_img.show()
```

Python中的图像处理——灰度化

- 灰度图像只包含亮度信息，无色彩
- Pillow中可以直接使用ImageOps.grayscale方法实现
- 常用于图像分析前的预处理
- 示例代码：

```
from PIL import ImageOps
gray = ImageOps.grayscale(img)
gray.show()
```

Python中的图像处理——提取红色通道

- 仅保留红色通道，设置G、B值为0
- 代码逻辑：遍历每个像素，手动调整RGB值

```
image = img.convert("RGB")
width, height = image.size
pixels = image.load()
for y in range(height):
    for x in range(width):
        r, g, b = pixels[x, y]
        img.putpixel((x, y), (r, 0, 0))
img.show()
```

Python中的图像处理——镜像翻转

- 图像的镜像翻转可以使用 `Pillow` 的 `transpose` 方法
- `Image.FLIP_LEFT_RIGHT` 可实现水平镜像
- 示例代码：

```
mirrored = img.transpose(Image.FLIP_LEFT_RIGHT)
mirrored.show()
```

Python中的图像处理——缩放与旋转

- 使用 `resize()` 实现缩放，`rotate()` 实现旋转
- `rotate(角度)` 会绕中心点顺时针旋转图像
- 示例代码：

```
img.resize((150, 150)).show()   # 缩小图像尺寸
img.rotate(45).show()   # 顺时针旋转45度
```

Python中的图像处理——绿幕替换与图像合成

- 利用绿色背景作为屏蔽色，将其替换为其他图像中对应像素
- 代码示例：

```
cat = Image.open("cat.jpg").resize((300, 300))
pixels = cat.load()
# 阈值判定背景区域：将亮度较高的区域替换为绿色
for y in range(cat):
    for x in range(cat.width):
        r, g, b = pixels[x, y]
        avg = (r + g + b) / 3
        if avg > 180: # 高亮度
            pixels[x, y] = (0, 255, 0)
```

Python中的图像处理——绿幕替换与图像合成

- 利用绿色背景作为屏蔽色，将其替换为其他图像中对应像素
- 代码示例：

```
pixels_bg = flower.load()
for y in range(cat.height):
    for x in range(cat.width):
        r, g, b = pixels_fg[x, y]
        avg = (r + g + b) / 3
        if g > avg * 1.5 and g > 150: # 绿色占主导，视为绿幕
            pixels_fg[x, y] = pixels_bg[x, y]
```

- Python中的动画制作与tkinter介绍
  - 动画是通过快速更新图像内容形成的视觉动态效果
  - 使用 `tkinter` 模块中的 `Canvas` 可以实现动画效果
  - 动画核心组成包括：
    - 创建画布：`canvas = Canvas(width, height)`
    - 创建形状：`canvas.create_oval(), create_rectangle()`
    - 控制移动：`canvas.move(shape, dx, dy)`
    - 实时刷新：`canvas.update() + time.sleep(`间隔`)`

Python中的动画制作与tkinter介绍

- 动画机制本质是一个持续的“心跳循环”：
  - 每次更新 → 暂停一段时间 → 再次更新 → 形成帧
- `tkinter` 是Python自带的GUI工具包，支持画布Canvas操作
  - 常用方法：`create_oval`，`move`，`update` 等
- 通过循环和时间延迟不断更新图像位置，实现动画效果

Python中的动画制作与tkinter介绍

- tkinter制作动画代码基本结构

```
import tkinter as tk
root = tk.Tk()
canvas = tk.Canvas(...)
shape = canvas.create_oval(...)
canvas.move(shape, dx, dy)
canvas.after(20, ...)
root.mainloop()
```

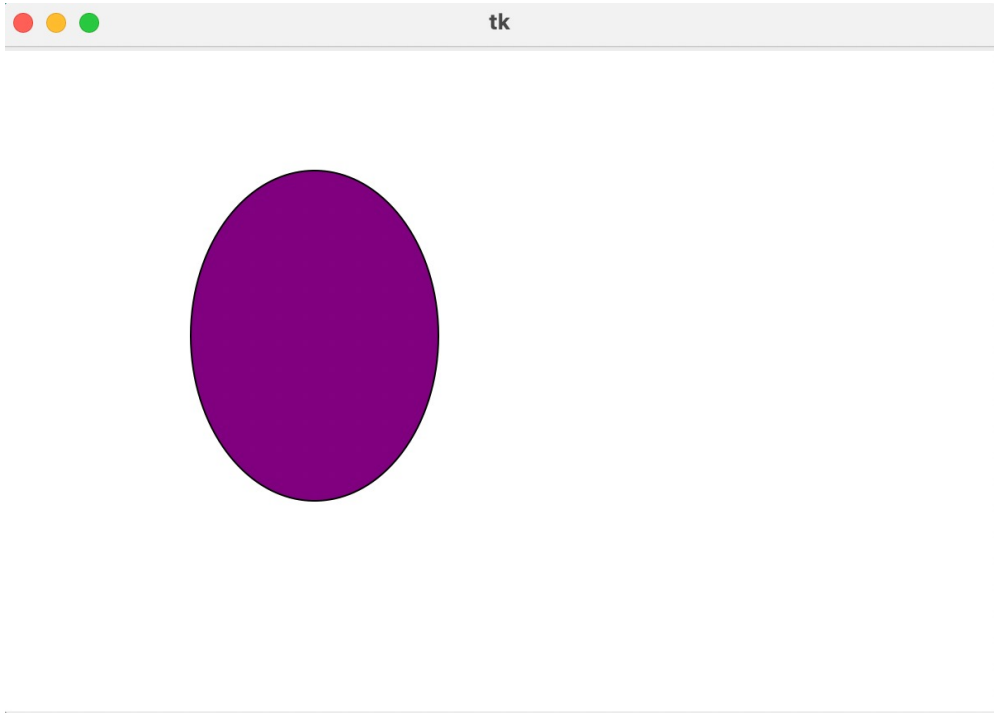
■ Python中的动画制作与tkinter介绍

- 动画制作举例：

```python
import tkinter as tk
import time
# 创建画布
root = tk.Tk()
canvas = tk.Canvas(root, width=600, height=400, bg='white')
canvas.pack()
# 创建图形
points = (50, 150, 200, 350)
shape = canvas.create_oval(*points, fill='purple')
# 移动速度
dx, dy = 5, 3
def animate():
    global dx, dy
    canvas.move(shape, dx, dy)
    x1, y1, x2, y2 = canvas.coords(shape)

    # 碰撞检测：左右边界
    if x2 >= 600 or x1 <= 0:
        dx = -dx
    # 上下边界
    if y2 >= 400 or y1 <= 0:
        dy = -dy

    canvas.after(20, animate)  # 每20ms调用一次自身（类似帧刷新）
animate()  # 启动动画
root.mainloop()  # 进入主事件循环
```



Python中的动画制作与tkinter介绍

- 举例：

```
import tkinter as tk # 导入 Tkinter 模块
root = tk.Tk()
```

- 创建主窗口对象 `root`，所有 `Tkinter` 程序的起点，相当于应用的容器窗口

Python中的动画制作与tkinter介绍

- 举例：

```
canvas = tk.Canvas(root, width=600, height=400, bg='white')
```

  - 在主窗口 `root` 中创建一个 **画布**，大小为 `600×400` 像素，背景为白色

```
canvas.pack()
```

  - 使用 `pack()` 方法将画布添加进窗口（否则不会显示）

Python中的动画制作与tkinter介绍

- 举例：

```
points = (50, 150, 200, 350)
```

- 定义一个四元组 `points`，表示椭圆外接矩形的左上角 `(50, 150)` 到右下角 `(200, 350)`

```
shape = canvas.create_oval(*points, fill='purple')
```

- 创建一个椭圆，颜色为紫色，存储为 `shape` 对象的 `ID`
- `*points` 解包，将四个值分别传入函数参数中

Python中的动画制作与tkinter介绍

- 举例：

```
dx, dy = 5, 3
```

  - 定义每帧中椭圆在水平方向（x轴）和垂直方向（y轴）移动的像素距离
  - 可以理解为“速度向量”

```
def animate():
```

  - 定义一个动画函数 `animate()`，该函数会在每一帧更新图形位置，并调用自身实现循环

- Python中的动画制作与tkinter介绍
  - 举例：

```
global dx, dy
```

    - 声明要在函数内使用函数外部的变量 `dx` 和 `dy`

```
canvas.move(shape, dx, dy)
```

    - 将图形 `shape` 沿当前速度向量 `(dx, dy)` 移动
    - 移动单位是像素

- Python中的动画制作与tkinter介绍
  - 举例：

```
x1, y1, x2, y2 = canvas.coords(shape)
```

    - 获取当前图形 `shape` 的边界框坐标
    - 返回的是 (左上x，左上y，右下x，右下y)
    - 用于判断是否碰到了边界

Python中的动画制作与tkinter介绍

- 举例：

```
if x2 >= 600 or x1 <= 0:
    dx = -dx
```

- 如果图形右侧超过画布右边界，或左侧小于0，就反向 dx
- 实现“水平反弹”

```
if y2 >= 400 or y1 <= 0:
    dy = -dy
```

- 同理，判断上下边界，实现“垂直反弹”

- Python中的动画制作与tkinter介绍
  - 举例：

```
canvas.after(20, animate)
```

    - 设置20毫秒后再次调用 `animate()`
    - 实现动画循环的关键，**不会阻塞窗口主线程**
    - 大约每秒50帧的刷新率（1000ms / 20ms ≈ 50fps）

- Python中的动画制作与tkinter介绍
  - 举例：

```
animate()
```

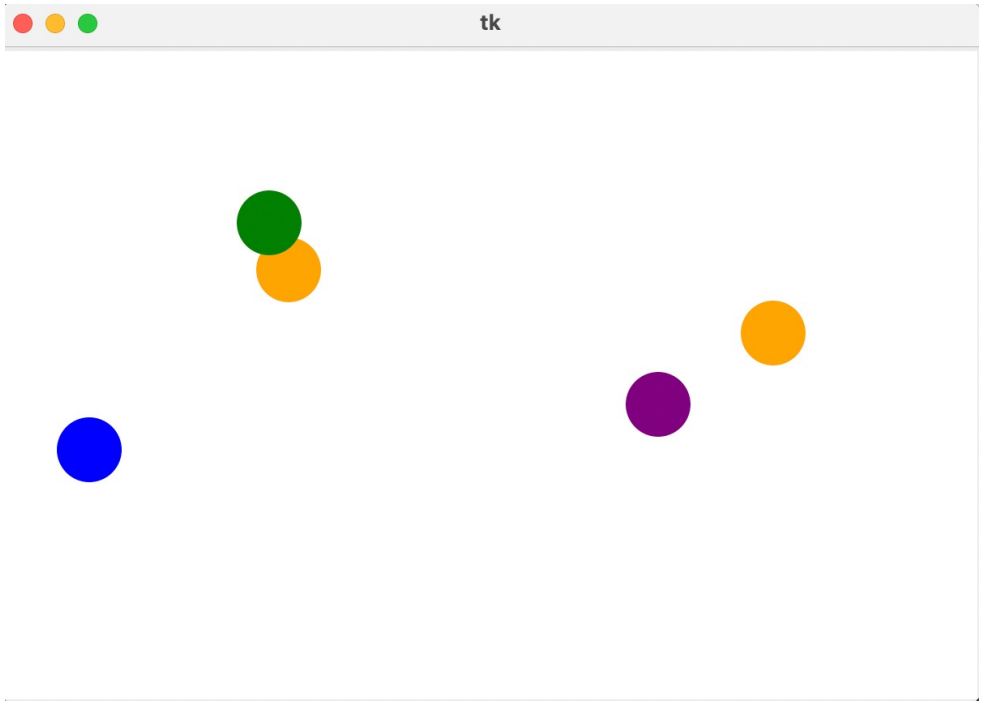
    - 启动动画循环，调用一次 `animate()` 进入定时递归

```
root.mainloop()
```

    - 启动 `Tkinter` 的主事件循环
    - 程序在此阻塞等待用户交互（窗口关闭、鼠标点击等）
    - 是 `GUI` 程序的核心运行入口

- Python中的动画制作与tkinter介绍
  - 拓展：多个小球碰撞动画

```python
# 随机创建多个小球
for _ in range(BALL_COUNT):
    x = random.randint(50, WIDTH - 50)
    y = random.randint(50, HEIGHT - 50)
    dx = random.choice([-4, -3, 3, 4])
    dy = random.choice([-4, -3, 3, 4])
    color = random.choice(['red', 'green', 'blue', 'purple', 'orange'])
    ball_id = canvas.create_oval(
        x - BALL_RADIUS, y - BALL_RADIUS,
        x + BALL_RADIUS, y + BALL_RADIUS,
        fill=color, outline=""
    )
    balls.append({'id': ball_id, 'dx': dx, 'dy': dy})
```



Python图像与动画总结：

- 图像处理部分：
  - 理解像素与RGB概念，掌握 `Pillow` 图像处理
  - 实现图像亮度调节、通道提取、镜像与合成

- 动画部分：
  - 学会使用 `tkinter` 创建画布与图形
  - 实现基本移动动画与弹跳逻辑

- 深入使用Python进行文本处理
  - 掌握字符串的高级使用方法和实用技巧
  - 掌握ASCII与字符编码的处理
  - 字符串的比较与排序操作
  - 应用场景：
    - 文本数据预处理与清洗
    - 信息抽取与分析
    - 输入验证与编码
    - 加密与解密

■ 字符串高级用法

- 由字符组成的有序序列，类型为str，不可变数据

方法	功能
`startswith(s)`	判断是否以某个子串开头
`endswith(s)`	判断是否以某个子串结尾
`title()`	所有单词首字母大写
`capitalize()`	首字母大写，其余小写
`zfill(n)`	左侧补零使字符串长度为 `n`
`center(n, char)`	居中对齐，用 `char` 填充到 `n` 长度

```
"python".zfill(8)          → '00python'
"hello".center(9, '-')     → '--hello--'
```

■ 字符串高级用法

- 分割与拼接操作
  - `split()` 和 `join()` 是文本预处理中的利器
  - `split()` 将字符串分割为列表
  - `join()` 将列表合并为字符串

```
text = "this is a test”
words = text.split()  # ['this', 'is', 'a', 'test’]
rejoin = "-".join(words)  # 'this-is-a-test'
```

■ 字符串高级用法

- 字符查找与替换

方法	功能
find(sub)	返回首次出现的位置
rfind(sub)	返回最后一次出现的位置

```
msg = "banana banana”
print(msg.replace("na", "NA"))   # baNANA banana
```

- 多用于文本模板替换、批量替换关键词

■ 字符串高级用法

- 大小写处理与标准化

方法	功能说明
`upper()`	全部转换为大写
`lower()`	全部转换为小写
`title()`	所有单词首字母大写
`swapcase()`	大小写互换

```
s = "hello WORLD"
print(s.swapcase())  # 'HELLO world'
```

- 多文本统一格式并帮助文本比较与分类

■ 字符串高级用法

- 字符串判断型方法（返回布尔值）

方法	功能
`isalpha()`	是否全为字母
`isdigit()`	是否全为数字
`isalnum()`	是否只含字母或数字
`isspace()`	是否只含空白字符

```
print("ABC123".isalnum())   # True
print("   ".isspace())      # True
```

字符串高级用法

- 字符与ASCII编码的转换
  - ASCII 是最基本的字符编码系统
    - ord(char)：字符 → 整数
    - chr(int)：整数 → 字符
  - `print(ord('A'))  # 65`
  - `print(chr(97))  # 'a'`

ASCII可显示字符（共95个）

二进制	十进制	十六进制	图形	二进制	十进制	十六进制	图形	二进制	十进制	十六进制	图形
0010 0000	32	20	(space)	0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p
0011 0001	49	31	1	0101 0001	81	51	Q	0111 0001	113	71	q
0011 0010	50	32	2	0101 0010	82	52	R	0111 0010	114	72	r
0011 0011	51	33	3	0101 0011	83	53	S	0111 0011	115	73	s
0011 0100	52	34	4	0101 0100	84	54	T	0111 0100	116	74	t
0011 0101	53	35	5	0101 0101	85	55	U	0111 0101	117	75	u
0011 0110	54	36	6	0101 0110	86	56	V	0111 0110	118	76	v
0011 0111	55	37	7	0101 0111	87	57	W	0111 0111	119	77	w
0011 1000	56	38	8	0101 1000	88	58	X	0111 1000	120	78	x
0011 1001	57	39	9	0101 1001	89	59	Y	0111 1001	121	79	y
0011 1010	58	3A	:	0101 1010	90	5A	Z	0111 1010	122	7A	z
0011 1011	59	3B	;	0101 1011	91	5B	[	0111 1011	123	7B	{
0011 1100	60	3C	<	0101 1100	92	5C	\	0111 1100	124	7C	|
0011 1101	61	3D	=	0101 1101	93	5D	]	0111 1101	125	7D	}
0011 1110	62	3E	>	0101 1110	94	5E	^	0111 1110	126	7E	~
0011 1111	63	3F	?	0101 1111	95	5F	_				

■ 字符串高级用法

- 字符编码与加密

```
def caesar_encrypt(text, shift):
    result = ''
    for ch in text:
        if ch.isalpha():
            base = ord('A') if ch.isupper() else ord('a')
            result += chr((ord(ch) - base + shift) % 26 + base)
        else:
            result += ch
    return result
```

- caesar_encrypt("Hello", 2) → Jgnnq

■ 字符串高级用法

- 字符串比较与排序
  - 字符串比较按字典序（lexicographic order）
  - 'A' < 'B' < 'Z' < 'a' < 'b'
  - 区分大小写，大写在前，按字母顺序比较

```
print("apple" < "banana")     # True
print("Zebra" < "apple")      # True
print("Abc" == "abc")         # False
```

  - 用于排序、搜索、词典处理等

■ 文本处理实战

- 文本反转处理：实现一个函数，反转输入的英文句子

```
def reverse_words(sentence):
    words = sentence.split()
    return ' '.join(reversed(words))


reverse_words("hello world python")
"python world hello"
```

文本处理实战

- 统计关键词频：输入文本，输出单词对应词频字典

```
def keyword_count(text):
    words = text.lower().split()
    return {w: words.count(w) for w in set(words)}

keyword_count("this is a test this is only a test")
{'this': 2, 'is': 2, 'a': 2, 'test': 2, 'only': 1}
```

- 可用于舆情分析、搜索推荐

文本处理实战

- 去除标点与清洗文本：输入文本，输出大小写标准化、去除标点的文本

```
import string
def clean_text(text):
    return text.translate(
        str.maketrans('', '', string.punctuation)
    ).lower()
clean_text("Hello, world!")
"hello world"
```

文本处理实战

- 检测email邮箱格式：

```
def is_valid_email(s):
    return "@" in s and "." in s and s.index('@') < s.rindex('.')
```

- s.rindex() # 字符串中最后出现某字符的index
- is_valid_email("abc@xyz.com") # True
- is_valid_email("abc@xyzcom") # False

文本处理实战

- 提取文件拓展名：

```
def get_extension(filename):
    parts = filename.split('.')
    return parts[-1] if len(parts) > 1 else ''
```

- get_extension("photo.jpg") → "jpg"
- get_extension("README") → ""

文本处理实战

- 判断回文字符串：
  - 正序和倒序完全一样（只考虑数字和字母）

```
def is_palindrome(s):
    clean = ''.join(c for c in s if c.isalnum()).lower()
    return clean == clean[::-1]
```

  - `is_palindrome("A man, a plan, a canal, Panama!")` → `True`

- 文本处理实战
  - 查找最长单词：

```
def longest_word(text):
    words = text.split()
    return max(words, key=len)
```

  - longest_word("a fox jumps over a lazy dog") → "jumps"
  - 适合文本结构分析

■ 文本处理实战

- 按单词长度分组:

```
def group_by_length(text):
    words = text.split()
    groups = {}
    for word in words:
        groups.setdefault(len(word), []).append(word)
    return groups
```

- group_by_length("the cat jumped over the fence")

→ {3: ["the", "cat"], 6: ["jumped"], 4: ["over"], 5: ["fence"]}

- 文本处理要点：
  - 常用字符串高级函数与编码技巧
  - 字符比较与排序规则
  - 面向应用的文本处理方案
  - 拓展练习：
    - 关键词高亮显示
    - 简易搜索匹配引擎
    - 用户名/邮箱/手机号验证器
    - 批量处理文件名与路径提取

- 嵌套数据结构的概念与使用
  - 掌握列表和字典嵌套使用的方法
  - 掌握嵌套数据结构的操作和应用
  - 示例：
    - 列表中包含字典
    - 字典中包含列表
    - 字典中包含字典

- 为什么要使用嵌套数据结构?
  - 可以有效组织复杂数据
  - 易于数据访问和管理（Python自带的各种方法）
  - 便于数据持久化存储（如JSON）
  - 符合实际应用中的多种场景（一切现实对象的表示）
  - 应用示例：
    - 地图API、用户信息、游戏状态等

嵌套列表

- List of lists

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2]) # 输出：6
```

  - matrix为二维数组（矩阵），其数组形状可以用(3, 3)表示
  - 对matrix中最细粒度元素的获取，即对每个维度逐一索引的过程

■ 嵌套字典

- Dictionary of dictionaries

```
student = {
  "name": "Anna",
  "scores": {"math": 90, "english": 85}
 }
print(student["scores"]["english"])   # 输出 85
```

- 字典的value可以是任意数据类型
- 获取value的过程是字典按key下钻的过程

■ 混合嵌套

- List of dictionaries

```
students = [
  {"name": "Tom", "score": 92},
  {"name": "Lucy", "score": 88}
]
print(students[0]["name"])   # 输出 Tom
```

- 常用于有顺序的现实事物集合

■ 混合嵌套

- Dictionary of lists

```
gradebook = {
  "Tom": [88, 92, 85],
  "Lucy": [90, 91, 89]
 }
print(gradebook["Lucy"][1])   # 输出 91
```

- 常用于单个事物具有按顺序的属性

■ 多层嵌套

- Dictionary of lists of dictionaries

```
data = {
    "students": [
        {"name": "Alice", "grades": {"math": 80, "cs": 90}},
        {"name": "Bob", "grades": {"math": 70, "cs": 85}}
    ]
}
print(data["students"][1]["grades"]["cs"])  # 输出 85
```

- 根据场景需求的不同可以设置任意复杂程度的嵌套数据结构

嵌套数据结构的遍历

- 遍历list of lists:

```
matrix = [[1, 2], [3, 4]]
for row in matrix:
    for item in row:
        print(item)
```

- 遍历dictionary of lists:

```
for name, scores in gradebook.items():
    print(name, max(scores))
```

嵌套数据的反转

- 将字典的key-value进行反转，形成嵌套结构：

```
ages = {"Tom": 20, "Jerry": 25, "Bob": 20}
reversed_dict = {}
for name, age in ages.items():
    reversed_dict.setdefault(age, []).append(name)
print(reversed_dict)
# 输出：{20: ['Tom', 'Bob'], 25: ['Jerry']}
```

- 用value做key时要注意多个key可能对应相同的value，用list保存

Nested Data

- JSON与Python嵌套结构的映射
  - JSON格式：以字符串格式存储的嵌套数据结构
  - 读取JSON数据为Python嵌套数据

```
import json
with open('data.json') as f:
    data = json.load(f)
with open('data.json') as f:
    json.dump(data, f)
```

Nested Data

- 嵌套数据结构的实例
  - 天气数据

```
weather = {
    "Monday": {"temp": 20, "humidity": 0.6},
    "Tuesday": {"temp": 22, "humidity": 0.7}
}
for day, info in weather.items():
    print(day, "温度: ", info["temp"])
```

- 嵌套数据结构的实例
  - 地图数据

```
markers = [
    {"name": "Park", "pos": [25.12, 55.15]},
    {"name": "Mall", "pos": [25.20, 55.27]}
]
for marker in markers:
    print(marker["name"], "位于: ", str(marker ["pos"]))
```

Nested Data

- 嵌套数据结构的实例
  - OpenAI大模型聊天数据

```
response =
result["choices"][0]["message"]
["content”]
```


```
{
  "model": "gpt-4o-2024-08-06",
  "request_id": "req_ded8ab984ec4bf840f37566c1011c417",
  "usage": {
    "total_tokens": 31,
    "completion_tokens": 18,
    "prompt_tokens": 13
  },
  "top_p": 1.0,
  "temperature": 1.0,
  "metadata": {},
  "choices": [
    {
      "index": 0,
      "message": {
        "content": "Hello! I'm ChatGPT, your assistant.",
        "role": "assistant"
      },
      "finish_reason": "stop"
    }
  ]
}
```

- 解决实际问题时往往面临效率上的考量
  - 系统的准确性是基本保障，效率影响的是用户体验，二者缺一不可
  - 搜索引擎每天处理数十亿条请求
  - AI大模型需要在毫秒内给出响应
  - 这么大的数据存下来要占用多少空间?
  - 多块的响应才是满足用户要求的?

- 正确性与效率的权衡
  - 正确性优先，但大数据处理或高频调用中效率至关重要
  - **示例**：基于递归的Fibonacci算法

```
# 正确但低效的递归实现
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

  - 结果正确但是分支爆炸

- 正确性与效率的权衡

```
memo = {}
def fast_fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fast_fib(n-1) + fast_fib(n-2)
    return memo[n]
```

- 使用记忆模块提高效率，空间换时间

■ 时间效率与空间效率

- 时间效率：运行快慢
- 空间效率：内存使用量
- **常见权衡方式**：缓存中间结果（空间换时间）、数据预处理（时间换空间）
- 在资源有限场景下，对于程序时间和空间的衡量至关重要
- 根据资源的特点选择权衡的方式

■ 衡量时间效率——计时模块

- Python内置 time 模块：测量一段代码执行前后的时间间隔

```
import time

start = time.time()

# 要测量的代码

end = time.time()

print("运行时间：", end - start, "秒")
```

  - time.time() 返回当前时间戳（1970以来的秒数）

- 衡量时间效率——计时模块
  - 有些代码耗时不随输入变化
    - 如：温度换算算法

```
def c_to_f(c):
    return c * 9.0 / 5 + 32
start = time.time()
c_to_f(37)
print("耗时：", time.time() - start)
```

    - 几乎感受不到耗时

- 衡量时间效率——计时模块
  - 有些代码耗时随输入的变化而变化
    - 如：累加函数

```
def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

    - 尝试不同输入：100, 1000, 10000
    - 耗时随输入值的增加而线性增长

■ 衡量时间效率——计时模块

- 有些代码耗时随输入的变化而变化
  - 如：基于加法的平方函数

```
def square(n):
    sqsum = 0
    for i in range(n):
        for j in range(n):
            sqsum += 1
    return sqsum
```

- 双层循环 → 随输入的增大耗时呈平方指数增长

- 衡量时间效率——操作计数
  - 计时并不完全稳定和准确
    - 不同电脑、系统、后台进程、语言实现均会影响时间结果
    - 计时适合初步评估，不适合精确比较
  - 是否有更加准确的评估程序效率的方式?
    - 操作计数
    - 忽略机器因素，只关注操作步骤数
    - 假设基本操作如加法、赋值视为常数时间
    - 主要分析控制结构（循环）
    - 优势：与机器无关，评估算法结构优劣

- 衡量时间效率——操作计数
  - 操作计数的示例
    - 温度换算算法：

```
def c_to_f(c):
    return c * 9.0 / 5 + 32
```

    - **操作数**：乘法（1），除法（1），加法（1）→ 总计3步
    - 无论输入值多少，操作数恒定

■ 衡量时间效率——操作计数

- 操作计数的示例
  - 累加函数：

```
def mysum(x):
    total = 0          # 1
    for i in range(x+1):   # x+1 次
        total += i     # x+1 次
    return total       # 1
```

  - **总操作数**：1 + (x+1)*2 + 1 = 2x + 4
  - 线性增长，符合O(n)　(时间复杂度与输入参数的关系为一次线性函数)

■ 衡量时间效率——操作计数

- 操作计数的示例
  - 基于加法的平方函数:

```
def square(n):
    sqsum = 0
    for i in range(n):
        for j in range(n):
            sqsum += 1
    return sqsum
```

- **总操作数**: n*n 次加法 = $O(n^2)$时间复杂度与输入参数的关系为二次指数函数

■ 计时与计数对比

特性	计时	计数
精度依赖	高，受环境影响	低，独立于系统
是否可复现	否	是
对算法评估	一般	精确
分析结构优势	弱	强

■ 算法复杂度——大O

- 操作计数提供定量分析
- 仍需要给出系统的增长趋势描述
- 关注省略常数系数与低阶项后的系统主要开销
- 算法复杂度分析：大O
- **常见复杂度**：$O(1)$, $O(n)$, $O(n^2)$, $O(\log n)$

■ 如何基于大O分析优化已有算法

- 原始的累加函数：

```
def mysum(x):
    total = 0          # 1
    for i in range(x+1):   # x+1 次
        total += i     # x+1 次
    return total       # 1
```

- 只关注高阶项：n次循环
- 算法复杂度：O(n)

- 如何基于大O分析优化已有算法
  - 改进后的累加函数：

```
def mysum_fast(n):
    return n * (n + 1) // 2
```

    - 循环的线性增长变为了一次计算结果（虽然有多个常数操作）
    - 复杂度变化：O(n) → O(1)　（有多个常数操作，可以忽略常数项）

■ 分析一段代码的复杂度

- 三重嵌套：

```
def triple_loop(n):
    count = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                count += 1
    return count
```

- **操作计数**：n^3 次加法 → 算法复杂度：$O(n^3)$

计时与计数总结

- 计时方法快速精准但不稳定
- 操作计数适合分析算法效率
- 大O复杂度帮助我们关注增长趋势
- 写代码时应兼顾正确性和效率

Reading and QA Time

See you next week !