中国海洋大学
OCEAN UNIVERSITY OF CHINA

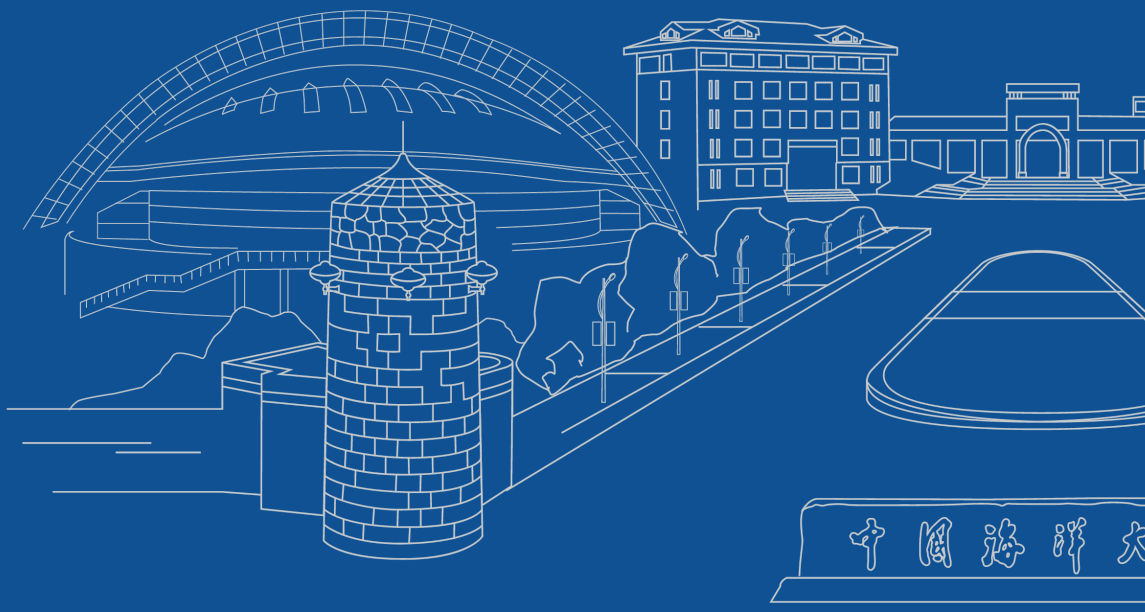
系统开发工具基础第三次实验报告

了解 Shell 环境　学习使用 Python 的视觉应用

于景一

23090032047

信息科学与工程学部
计算机学院
计算机科学与技术专业



OUC, September 2024

中国海洋大学
OCEAN UNIVERSITY OF CHINA

系统开发工具基础第三次实验报告

了解 Shell 环境　学习使用 Python 的视觉应用

于景一

23090032047

指导老师: 周小伟
系统开发工具基础, 计算机学院

信息科学与工程学部
计算机学院
计算机科学与技术专业

OUC, September 2024

概述

本实验报告使用由**于景一**制作的LaTeX模板完成。关于此模板的信息，您可以前往GitHub模板仓库具体了解。[1][2]本文章使用的是“实验报告”模板。[3]

本实验报告是*系统开发工具基础课程*的第三次实验报告，主要关于 **Shell 环境（进程管理）与 Python 的语言基础、在计算机视觉方面的应用**，总结实战经验，记录心得体会。

对于 **Shell 的进程管理**方面，由于我早先有过使用 `screen` 的经验，所以对此部分内容并不陌生。在本次实验中，我们将深刻理解 **Shell 中的任务控制的机理**，了解*信号机制的通信原理*，学会*结束进程、挂起进程、后台执行进程*，并且掌握除 `screen` 外的*终端多路复用的方法*（即 `tmux`）；由于我拥有许多云服务器，需要日常进行运维，对**远端设备的操作（SSH）**已经熟稔，在文中仅简略概括。

至于 **Python 方面的基础语法与计算机视觉的应用**，由于先前我有使用多线程技术编写过一些 Python 小程序，并在上周的数学建模国赛中使用过许多数学计算与图形绘制的相关库，如 `Matplotlib`，`Scipy`，`Numpy` 等，所以对此部分内容也早有涉猎。在本次实验中，我们将首先梳理 **Python 的基础语法**，将它与*系统学习过的 C/C++ 进行横向对比学习*，尔后**学习 Python 的视觉应用**，了解图像处理的基本操作，如图像的读取、显示、保存、基本的图像处理这些最基本的操作（参考Solem, 2014），并且**额外地**尝试编写一些简单的视觉应用，如图像的灰度化、二值化、边缘检测、图像的腐蚀与膨胀等。

NOTE

知识之海是无边无际的，只是尽力游弋，就已倍感费时费力，然而学习的过程是美好的。本次实验在 `Shell` 和 `Python` 方面的粗浅研究，仅可窥得其冰山一角。本文十分惭愧地呈现了我在实验中对两个领域的浅薄了解，如有谬误还请批评斧正。

[1] 或您可直接搜索 GitHub 账号 *@jstar0* 了解更多

[2] 您请注意，本模板基于 LPPL v1.3c 分发，本项目在原模板Polytechnic University of Leiria: LaTeX Thesis Template的基础上进行了合法地大量二改，包括但不限于自定义风格、中文化支持、样式重定义、功能增加等。

[3] 模板提供两种样式，一种为学术论文样式，另一种为实验报告样式，具体区别请检查 GitHub 仓库上的两个分支。

目录

1

Shell 环境管理

1.1 Shell 中的任务控制

1.1.1 概述（前台任务与后台任务）

我们需要首先明确，在 Shell 中，**进程管理包括前台任务和后台任务的控制**。每个进程都可以在前台运行，接受用户的直接输入和输出，或在后台运行，不影响用户继续在终端执行其他命令。

所谓**前台任务**，即 *Foreground Job*，就是指一个进程在前台运行时，用户会等待它执行完成后才能继续输入命令。类似的，**后台任务** *Background Job*，就是进程在后台运行时，不会占用 Shell 的控制权，用户可以继续操作其他任务。

我们根据定义可以知晓，其实所谓前、后台，都是 *Shell* 相对于用户的一个概念，即用户是否能够直接与进程进行交互，如能，则为前台任务，否则为后台任务。

有了上述的概念，我们就可以开始探讨**如何在 Shell 中控制任务**了。

1.1.2 任务控制的基本操作

首先我们需要明确，在 Shell 中**任务共有三种状态**，即运行、挂起、终止。我们通过一些常用的指令来控制任务的状态。

- `Ctrl + Z`：将当前任务挂起（即将任务转为后台任务）。
- `Ctrl + C`：终止当前任务。
- `Ctrl + D`：终止当前 Shell 会话。
- `bg`：将任务转为后台任务。
- `fg`：将任务转为前台任务。
- `jobs`：查看当前 Shell 中的任务。

下面我们给出一些示例，在我的 `Ubuntu Azure SG` 服务器上进行测试，来具体化解释这些指令的用法。

第一组示例，我们将示范如何将单个任务转为后台任务[1]，见代码 1.1。

[1] 至于为什么强调单个，因为如果有多个进程须指定任务号。如果只有一个进程，则不需指定任务号（除非使用 `kill`）

```
wget -c --limit-rate=10k https://releases.ubuntu.com/noble/ubuntu-24.04.1-live-server-amd64.iso # 以 -c 允许断点续传，以 --limit-rate=10k 限速下载

# 此时任务在前台运行，我们可以按下 Ctrl + Z 将任务挂起
^D # Ctrl + Z 将任务挂起
# 将提示 [1]+  Stopped  wget -c --limit-rate=10k https://... 任务已经被挂起

bg # 将任务转为后台任务
# 此时控制台提示 [1]+ wget -c --limit-rate=10k https://... & 任务已经转为后台任务
# 并提示 Redirecting output to ‘wget-log’.

jobs # 使用 jobs 验证之
# 此时控制台提示 [1]+  Running  wget -c --limit-rate=10k https://... &

# 若要在此时直接终止进程，使用：
kill %1 # 杀死后台任务

# 或者将任务转为前台任务，再终止：
fg # 可以将任务转为前台任务
# 此时任务又转为前台任务
^C #Ctrl + C 以终止任务
```

代码 1.1: 任务控制实例一：以后台任务运行

第二组示例，我们将示范如何同时对多个进程进行任务控制，见代码 1.2。

```
sleep 1000 & # 启动一个后台任务，其中 & 表示后台运行
# 此时控制台提示 [1] 629227 任务号和 PID 号
sleep 2333 & # 再启动一个后台任务
# 此时控制台提示 [2] 629228 任务号和 PID 号
sleep 1234 & # 再启动一个后台任务
# 此时控制台提示 [3] 629229 任务号和 PID 号
jobs
# 控制台返回
# [1]   Running                 sleep 1000 &
# [2]-  Running                 sleep 2333 &
# [3]+  Running                 sleep 1234 &
# 特别的，任务号后的 + 表示当前任务，- 表示上一个任务，无符号表示其他任务
# 这些任务都在并行运行，通过
ps -ef | grep sleep
# JStar0   629227  629216  0 15:35 pts/1    00:00:00 sleep 1000
# JStar0   629228  629216  0 15:35 pts/1    00:00:00 sleep 2333
# JStar0   629229  629216  0 15:35 pts/1    00:00:00 sleep 1234
# 可以看到这三个任务都在后台运行

fg %2 # 将任务 2 转为前台任务
# 此时返回 sleep 2333 任务转为前台任务
^C # Ctrl + C 以终止任务

kill %1 %3 # 杀死任务 1 和任务 3
# 此时返回
# [1]-  Terminated              sleep 1000
# [3]+  Terminated              sleep 1234

jobs
# 此时不返回任何任务，所有任务都已经终止
```

代码 1.2: 任务控制实例二：对多个任务进行控制

1.2 深入探索信号机制

1.2.1 信号在进程管理的概念

类 Unix 系统中，信号是一种**软件中断**，允许操作系统和进程之间进行通信。信号的发送者，可以是操作系统、硬件或用户，将给接收者即进程，告知执行特定的操作或响应某个事件。Shell 中的进程管理**通过信号来控制**前台和后台任务。

参考Wikipedia contributors, 2024，在数字计算机中，中断 *an interrupt* (*sometimes referred to as a trap*) 是对处理器的请求，以中断当前正在执行的代码（在允许的情况下），以便及时处理事件。如果请求被接受，处理器将暂停其当前活动，保存其状态，并执行称为中断处理程序（或中断服务例程，*ISR*）的函数来处理事件。这种中断通常是临时的，在中断处理程序完成后允许软件恢复正常活动，尽管中断也可能表示致命错误。

1.2.2 常见的信号

信号是有限的消息，用于通知进程发生了某些事件。**每种信号都有一个编号和名称**，操作系统会根据事件类型发送相应的信号给进程。常见的信号如下：

信号	描述
SIGINT (信号 2)	中断信号。当用户按下 Ctrl + C 时，系统向前台进程发送这个信号，通常用于停止进程。
SIGKILL (信号 9)	强制终止信号。无论进程是否忽略信号，SIGKILL 都会立即终止进程。不能被捕捉或忽略。
SIGTERM (信号 15)	终止信号，表示请求进程优雅地退出。进程可以捕捉这个信号并在退出前执行清理操作。
SIGHUP (信号 1)	挂起信号，当终端关闭时，系统会发送这个信号，通常用于告诉守护进程重新读取配置文件。
SIGSTOP (信号 19)	停止进程。不能被捕捉或忽略，和 SIGKILL 类似，但不会终止进程，只是暂停。
SIGCONT (信号 18)	继续信号，用于恢复被暂停的进程。

一般我们通过相关的命令和快捷键来发送信号，如 `kill` 命令，将发送信号 `SIGTERM` 给指定的进程。具体的实现细节，在使用命令和快捷键时，我们不需要关心，只需要知道**信号是一种进程间通信的方式**即可。

类似的，我们也可以**给出其他的对应关系**。如 `Ctrl + C -> SIGINT`，终止前台任务；`Ctrl + Z -> SIGTSTP`，将前台任务挂起（转为后台停止状态）；`bg` 发送 `SIGCONT`，恢复挂起的任务在后台运行等。

使用 Python 中内置的 `signal` 库，我们也可以**捕捉信号**，如代码 1.3所示。

```python
import signal, time

def signalHandler(signum, time):
    print("\n获取到信号: ", signum)

signal.signal(signal.SIGINT, signalHandler)
i = 0
```

```
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

代码 1.3: *Python 模拟的信号捕捉*

那么，此程序在运行时，不会因为按下 `Ctrl + C` 而终止，而是会输出获取到信号：2，即捕捉到了 `SIGINT` 信号。

这个程序的运行结果如Figure 1.1所示。

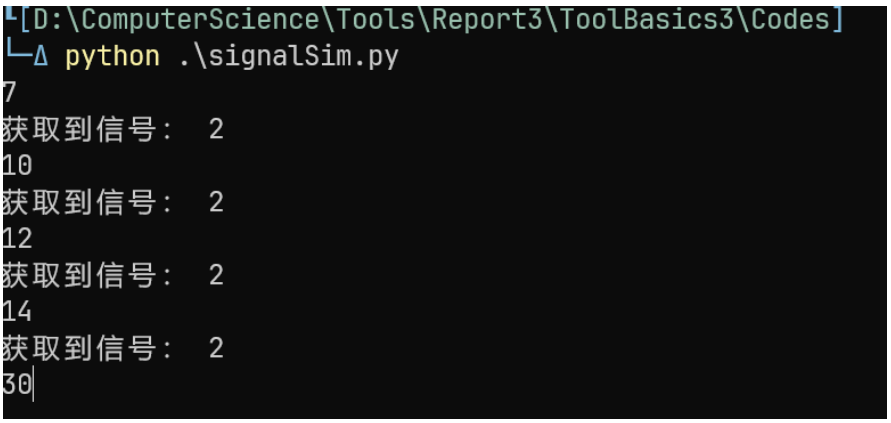


图 1.1: *Python 模拟的信号捕捉*

1.3 Shell 中的终端多路复用

我们主要比较两种常见的终端多路复用工具，即 `screen` 和 `tmux`。

1.3.1 功能比较

我们给出一个简单的功能比较表，见Table 1.2所示。

功能/特性	tmux	screen
会话管理	可以命名、分离、重新连接和恢复多个会话	同样支持会话的分离和恢复
窗口布局	提供灵活的窗口布局，可以分屏（水平/垂直）	支持分屏，但功能较基础
滚动缓冲区	默认支持滚动缓冲区，使用快捷键轻松滚动	需要手动启用滚动缓冲区，操作较繁琐
配置文件	配置文件位于 /.tmux.conf，支持更复杂的配置	配置文件位于 /.screenrc，配置功能较少
会话共享	支持会话共享，允许多个用户连接同一个会话	同样支持会话共享
插件扩展	支持插件扩展，可以通过 tmux 插件管理器安装扩展	不支持插件扩展
状态栏	可高度自定义状态栏	状态栏不可定制
复制模式	内置复制模式，使用更简单	支持复制模式，但需要额外配置
热键风格	默认热键风格现代，易于使用	热键风格传统，学习成本稍高

表 1.2: *tmux 与 screen 功能比较*

1.3.2 指令上的比较

基于我对 screen 的认识和对 tmux 的学习，给出一个简单的指令比较表，如Table 1.3所示。

功能/操作	tmux 指令	screen 指令
启动新会话	tmux new -s <session_name>	screen -S <session_name>
分离会话	Ctrl + b 然后按 d	Ctrl + a 然后按 d
重新连接会话	tmux attach -t <session_name>	screen -r <session_name>
列出所有会话	tmux ls	screen -ls
关闭会话	tmux kill-session -t <session_name>	screen -S <session_name> -X quit
创建新窗口	Ctrl + b 然后按 c	Ctrl + a 然后按 c
在窗口间切换	Ctrl + b 然后按数字键 (0-9)	Ctrl + a 然后按数字键 (0-9)
关闭当前窗口	exit 或 Ctrl + b 然后按 &	exit 或 Ctrl + a 然后按 K
水平分屏	Ctrl + b 然后按%	Ctrl + a 然后按 ‘
垂直分屏	Ctrl + b 然后按”	Ctrl + a 然后按 S
在面板间切换	Ctrl + b 然后按箭头键	Ctrl + a 然后按 Tab
调整面板大小	Ctrl + b 然后按: 输入 resize-pane	Ctrl + a 然后按 Ctrl + 箭头键
重命名窗口	Ctrl + b 然后按,	Ctrl + a 然后按 A
重命名会话	tmux rename-session -t <old_name> <new_name>	不支持
会话共享	tmux attach -t <session_name>	screen -x <session_name>

表 1.3: *tmux 与 screen 指令比较*

1.3.3 实机测试

在实机上对我新学习的 tmux 进行了简单的测试，如代码 1.4所示。

```
tmux new -s test # 创建一个名为 test 的会话
# 此时会话已经创建，可以在其中执行命令

# 假设执行了一些命令，现在需要分离会话
# Ctrl + b 然后按 d 分离会话

tmux ls # 列出所有会话
# 此时会列出所有会话，包括 test

tmux attach -t test # 重新连接 test 会话
# 此时会重新连接到 test 会话

# 假设执行了一些命令，现在需要关闭会话

exit # 退出会话
# 此时会话已经关闭

tmux ls # 再次列出所有会话
# 此时会列出所有会话，不再包括 test
```

代码 1.4: *tmux 实机测试*

1.4 远程设备的操作

使用 SSH 连接到远程设备，是我们用来控制服务器、远程调试开发设备等的重要手段。 *SSH 就是 Secure Shell Protocol*，是一种加密的网络协议，即使在不安全的网络上也能够安全地运行网络服务。

下面给出一个简单的 SSH 连接示例，是我通过 OpenSSH 连接到我的 Azure SG 服务器的过程，如图Figure 1.2所示。

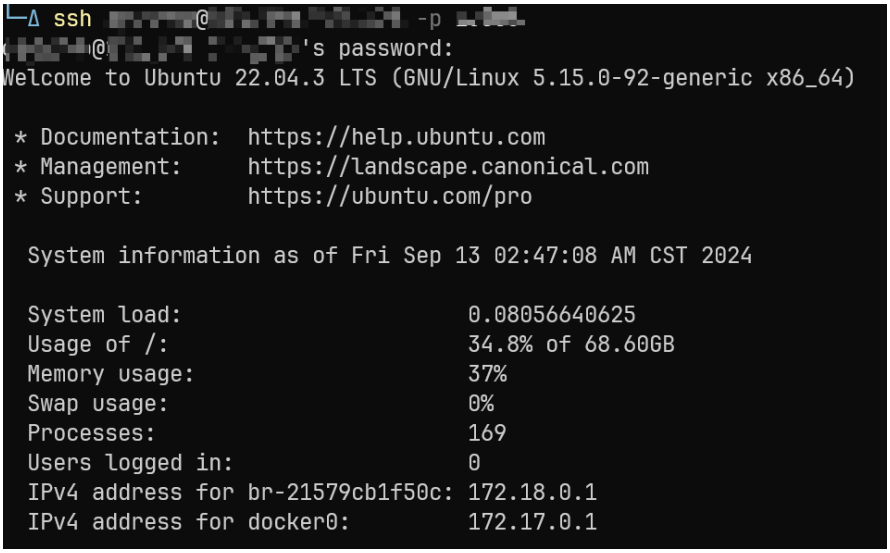


图 1.2: 终端的 *SSH* 示例

当然，我们还可以使用更先进的带 GUI 的 SSH 客户端，如我常用的 `termius`，如图Figure 1.3所示。

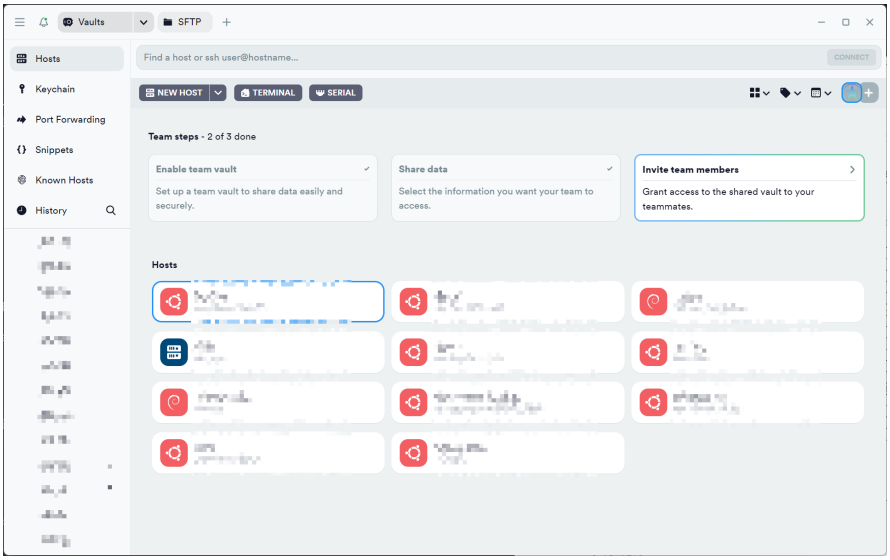


图 1.3: 使用 *Termius* 连接示例

1.5 小结

本章，我们首先**介绍了 Shell 中的任务控制**，包括前台任务和后台任务的概念，以及如何通过一些常用的指令来控制任务的状态。然后我们**深入探讨了信号机制**，介绍了信号的概念和常见的信号。最后我们**比较了两种常见的终端多路复用工具** `tmux` 和 `screen`，并给出了功能比较和指令比较。最后我们简单介绍了**远程设备的操作**，即通过 *SSH* 连接到远程设备。

2

Python 基础语法

2.1 Python 极简介绍

Python 是一门**通用型高级编程语言**，其设计哲学强调代码的可读性和简洁性，使用缩进来划分代码块。支持多种编程范式，包括面向对象、命令式和函数式编程。拥有动态类型系统和垃圾回收功能，能够轻松实现复杂的任务。*Python* 的标准库和第三方库生态系统非常丰富，使其成为科学计算、人工智能、Web 开发等领域的热门语言选择。简单易学而功能强大的特性，使 Python 成为编程初学者和专业开发者的共同选择。

2.2 Python 基础语法

有鉴于我们经过一学年的修习，对 C/C++ 的用法已经有了一定的了解，因此我们在这里不再从头开始讲解 Python 的基础语法，而是通过建立表格进行比较，以期更好地理解 Python 的语法特性。

具体的对比请见Table 2.1所示。

概念/语法	C/C++	Python	说明
变量声明	`int x = 10;`	`x = 10`	Python 不需要显式声明类型，解释器根据值类型推断。
注释	`// 单行注释` `/* 多行注释 */`	`# 单行注释` `""" 多行注释"""`	C/ C+ + 使用//和/ * */，Python 用 #或""" """进行注释。
打印输出	`printf("Hello, World! \n");`	`print("Hello, World!")`	C/ C+ + 使用 printf，Python 用 print。
条件语句	`if (x > 10) { ... } else { ... }`	`if x > 10: ... else: ...`	Python 用缩进代替大括号，且不需要括号包裹条件表达式。
循环语句	`for (int i = 0; i < 10; i++) { ... }` `while (x) { }`	`for i in range(10): ...` `while x:`	Python 的 for 循环直接使用 range() 函数，语法更加简洁。
函数定义	`int add(int a, int b) { return a + b; }`	`def add(a, b): return a + b`	Python 使用 def 定义函数，且无需声明返回类型。

数组/列表	`int arr[5] = {1, 2, 3, 4, 5};`	`arr = [1, 2, 3, 4, 5]`	Python 使用列表（`list`），可以动态调整大小，C/C++ 数组大小固定。
指针/引用	`int* ptr = &x;` `int& ref = x;`	不适用	Python 不支持指针，变量为引用类型，简化了内存管理。
内存管理	`malloc/ free` 或 `new/ delete`	由解释器自动管理内存	Python 有垃圾回收机制，C/C++ 需手动管理内存。
结构体/类	`struct Point { int x; int y; };` `class`	`class Point:` `def __init__(self, x, y): ...`	C 使用 `struct`，C++ 用 `class`，Python 类定义简单，使用 `self` 指代实例。
模块导入	`#include <stdio.h>`	`import math`	C/C++ 通过 `#include` 导入头文件，Python 使用 `import` 导入模块。
异常处理	不适用（C）；`try/catch`（C++）	`try: ... except: ...`	C++ 与 Python 都有异常处理机制，但 Python 语法更简单。
字符串处理	`char str[] = "Hello";` `std::string str = "Hello";`	`str = "Hello"`	C 字符串是字符数组，C++ 使用 `std::string`，Python 直接使用字符串。
类型转换	`(int)x` 或 `static_cast<int>(x)`	`int(x)`	C/C++ 需要强制类型转换，Python 用函数进行转换。
文件操作	`FILE *fp = fopen("file.txt", "r"); ... fclose(fp);`	`with open("file.txt", "r") as f: ...`	Python 使用 `with` 管理文件，自动处理关闭操作，C 需要手动关闭文件。
命令行参数	`int main(int argc, char* argv[])`	`import sys; sys.argv`	C 通过 `argc` 和 `argv[]` 获取参数，Python 通过 `sys.argv` 获取命令行参数。
库支持	需要手动链接库	内置大量标准库，且通过 `pip` 可安装第三方库	Python 库管理简化，无需手动链接，且有丰富的第三方库。

表 2.1: *Python 与 C/C++ 基础语法对比*

2.3 Python 语法示例

2.3.1 最佳实践

我们按照 *PEP 8* 风格指南，合理处理空格和缩进；建立入口点检查，即在 `if __name__ == "__main__"` 才执行主函数；使用函数式编程，避免全局变量；添加文档字符串，方便代码阅读和维护；在 `get_user_input` 函数中，使用 `try/except` 处理异常；变量命名上，使用 `snake_case` 风格。完整代码如代码 2.1所示。

```
#!/usr/bin/env python3
"""
Python 单文件的最佳实践示例

功能：
1. 读取用户输入的数字
2. 计算并输出数字的平方
```

```
"""

def get_user_input(prompt):
    """
    获取用户输入并将其转换为浮点数

    参数:
    prompt (str): 提示用户输入的消息

    返回:
    float: 用户输入的数字
    """
    while True:
        try:
            user_input = input(prompt)
            return float(user_input)
        except ValueError:
            print(" 输入无效，请输入一个有效的数字。")


def calculate_square(number):
    """
    计算给定数字的平方

    参数:
    number (float): 要计算平方的数字

    返回:
    float: 输入数字的平方
    """
    return number ** 2


def main():
    """
    主函数，执行脚本的主要功能
    """
    print(" 欢迎使用数字平方计算器！")
    number = get_user_input(" 请输入一个数字： ")
    squared_value = calculate_square(number)
    print(f"{number} 的平方是 {squared_value}")


if __name__ == "__main__":
    main()

```

代码 2.1: *Python* 最佳实践

2.3.2 多文件、进程和线程

Python 的多文件结构

我们都清楚，**在大型项目中，代码往往会被分成许多个文件**。而 Python 的设定就是，每个文件都可以作为模块导入，通过 `import` 语句实现。

下面给出一个实例，假使我们有 `math_tool.py` 和 `main.py` 两个文件：

```
# math_tool.py
def add(a, b):
```

```
    return a + b

def subtract(a, b):
    return a - b

# main.py
import math_tool

def main():
    x, y = 10, 5
    print(f"{x} + {y} = {math_tool.add(x, y)}")
    print(f"{x} - {y} = {math_tool.subtract(x, y)}")

if __name__ == "__main__":
    main()
```

代码 2.2: *Python 多文件结构示例*

在 `main.py` 中，我们通过 `import math_tool` 导入了 `math_tool.py` 文件，然后就可以使用其中的函数了。

Python 的进程

对于 **CPU 密集型任务，使用多进程**，就可以通过在操作系统中启动多个进程来实现并发执行。每个进程拥有独立的内存空间。Python 提供了 `multiprocessing` 模块来支持多进程编程。

我们给出一个示例，启动 5 个进程，每个进程执行 `worker` 函数，独立地并行处理任务。

```
from multiprocessing import Process

def worker(number):
    """ 模拟一些任务"""
    print(f"Worker {number} is working")

if __name__ == "__main__":
    processes = []
    for i in range(5):
        p = Process(target=worker, args=(i,))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()  # 等待所有进程完成

    print("Workers all done")
```

代码 2.3: *Python 多进程示例*

Python 的线程

类似于进程，对于 **I/O 密集型任务，使用多线程**，可以通过在同一进程中启动多个线程来实现并发执行。线程共享同一进程的内存空间。Python 提供了 `threading` 模块来支持多线程编程。

```
import threading
import time
```

```
3
4  def worker(number):
5      """ 模拟一些任务"""
6      print(f"Worker {number} is starting")
7      time.sleep(2)
8      print(f"Worker {number} is done")
9
10 if __name__ == "__main__":
11     threads = []
12     for i in range(5):
13         t = threading.Thread(target=worker, args=(i,))
14         threads.append(t)
15         t.start()
16
17     for t in threads:
18         t.join()  # 等待所有线程完成
19
20     print("Workers all done")
21
```

代码 2.4: *Python 多线程示例*

脚注

有必要具体展开讨论 **CPU 密集型任务和 I/O 密集型任务**为何分别推荐使用多进程和多线程。我们说：

使用**进程**，则每个进程都有独立的内存空间，进程间通信成本高，因此更适合 CPU 密集型任务。使用**线程**，则线程共享内存空间，开销较低，但容易出现竞争条件，因此适合 I/O 密集型任务。

实际上，我们还可以通过 `concurrent.futures` 模块来更方便地实现线程池和进程池。或是仅仅建立一个数组或列表，作为一个简易的线程池或进程池，然后通过 `join()` 方法等待所有线程或进程完成。

3

Python 图像处理

参考文献

Solem, J.E. (2014). *Python 计算机视觉编程*. 图灵程序设计丛书. 人民邮电出版社. ISBN: 9787115352323. URL: https://books.google.com.sg/books?id=-KQkswEACAAJ.

Wikipedia contributors (2024). *Interrupt — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Interrupt&oldid=1243602968. [Online; accessed 12-September-2024].

4

Annex A

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.