

## ◆ نکات عملکردی و ایمنی در کار با فایل‌ها

هنگام کار با فایل‌ها، رعایت نکات ایمنی و عملکردی مهم است تا از دست رفتن داده‌ها، خراب شدن فایل، و بروز مشکلات امنیتی جلوگیری شود. در این بخش، بهترین روش‌ها را برای نوشتن، خواندن، و محافظت از داده‌ها بررسی می‌کنیم.

### 1. جلوگیری از دست رفتن داده‌ها در هنگام نوشتن در فایل

هنگام نوشتن در فایل، باید از از بین رفتن داده‌های قبلی، خراب شدن اطلاعات، یا قطع شدن ناگهانی فرایند جلوگیری کرد.

#### ✓ 1.1 استفاده از حالت `a` به جای `w` برای جلوگیری از حذف داده‌های قبلی

در حالت `"w"`، اگر فایل از قبل وجود داشته باشد، تمام محتوای قبلی آن پاک می‌شود! برای افزودن اطلاعات بدون حذف داده‌های قبلی از حالت `"a"` استفاده کنید:

```
with open("log.txt", "a") as file:  
    file.write("\nورودی جدید به فایل اضافه شد")
```

✓ نتیجه:

- داده‌های قبلی حذف نمی‌شوند.
  - اطلاعات جدید به انتهای فایل اضافه می‌شوند.
- ✗ اگر حالت `"w"` استفاده شود، اطلاعات قبلی حذف می‌شوند!

#### ✓ 1.2 نوشتن در فایل موقت و جایگزینی امن

یک روش حرفه‌ای برای جلوگیری از خراب شدن فایل اصلی، نوشتن داده‌ها در یک فایل موقت و جایگزینی آن است.

```
import os  
  
file_path = "data.txt"  
temp_path = file_path + ".tmp"  
  
# نوشتن در فایل موقت  
with open(temp_path, "w") as temp_file:  
    temp_file.write("\nاطلاعات جدید")  
  
# جایگزینی فایل اصلی با فایل موقت  
os.replace(temp_path, file_path)
```

✓ مزایا:

- اگر هنگام نوشتن خطایی رخ دهد، فایل اصلی سالم باقی می‌ماند.
- از خراب شدن و از دست رفتن اطلاعات جلوگیری می‌شود.

## 2. بررسی صحت داده‌ها قبل از نوشتن در فایل




قبل از ذخیره کردن داده‌ها، باید بررسی کنیم که معتبر باشند. این کار از ورود اطلاعات نادرست یا مخرب جلوگیری می‌کند.

### 2.1 بررسی داده‌ها قبل از نوشتن

مثال: ذخیره کردن نام و سن در فایل، اما فقط اگر مقدار صحیح باشد.

```
def save_user_data(name, age):
    if not name.strip():
        print("خطا: نام نمی‌تواند خالی باشد")
        return
    if not isinstance(age, int) or age <= 0:
        print("خطا: سن باید یک عدد مثبت باشد")
        return

    with open("users.txt", "a") as file:
        file.write(f"{name}, {age}\n")
    print("داده‌ها ذخیره شدند.")

# مثال‌های معتبر و نامعتبر
save_user_data("محمد", 25) # ذخیره می‌شود 
save_user_data("", 30) # خطا: نام خالی است 
save_user_data("علی", -5) # خطا: سن نامعتبر است 
```

 مزیت: قبل از ذخیره شدن داده، بررسی‌های لازم انجام می‌شود تا از خراب شدن فایل جلوگیری شود.

## 3. استفاده از قفل فایل برای جلوگیری از تداخل در پردازش موازی

اگر چندین برنامه به‌طور همزمان به یک فایل دسترسی داشته باشند، ممکن است تداخل و از دست رفتن داده‌ها رخ دهد. برای جلوگیری از این مشکل می‌توان از قفل فایل (file locking) استفاده کرد.

### 3.1 قفل کردن فایل برای جلوگیری از نوشتن همزمان

```
from filelock import FileLock

lock = FileLock("data.txt.lock") # ایجاد قفل

with lock: # قفل کردن فایل
    with open("data.txt", "a") as file:
        file.write("داده جدید اضافه شد.\n")
```

 مزایا:

- از نوشتن همزمان چندین پردازش روی یک فایل جلوگیری می‌شود.
- از خراب شدن اطلاعات در فایل جلوگیری می‌شود.

## 4. جلوگیری از حملات امنیتی هنگام خواندن و نوشتن فایل

برخی حملات امنیتی می‌توانند اطلاعات را دستکاری کنند یا به اطلاعات حساس دسترسی داشته باشند. برای جلوگیری از این مشکلات باید موارد زیر رعایت شود.

### 4.1 جلوگیری از نوشتن فایل در مسیرهای حساس ✓

هرگز اجازه ندهید ورودی‌های کاربر مسیر فایل را تعیین کنند، زیرا ممکن است باعث دستکاری سیستم شود.

✗ کد ناامن:

```
filename = input("نام فایل را وارد کنید: ") # خطرناک!  
with open(filename, "w") as file:  
    file.write("اطلاعات حساس ذخیره شد.")
```

✓ کد امن:

```
import os  
  
filename = input("نام فایل را وارد کنید: ")  
safe_path = os.path.join("user_files", os.path.basename(filename)) # محدود به یک فولدر خاص  
with open(safe_path, "w") as file:  
    file.write("اطلاعات ذخیره شد.")
```

✓ مزیت: این روش مانع از دسترسی غیرمجاز به فایل‌های مهم سیستم می‌شود.

### 4.2 جلوگیری از خواندن اطلاعات حساس ✓

✗ کد ناامن:

```
with open("/etc/passwd", "r") as file: # ممکن است شامل اطلاعات حساس باشد  
    print(file.read())
```

✓ راه‌حل: فقط مسیرهای مجاز را اجازه دهید.

## 5. بهینه‌سازی عملکرد خواندن و نوشتن فایل‌ها

اگر فایل‌های بزرگ دارید، خواندن یا نوشتن کل فایل در یک مرحله ممکن است باعث مصرف زیاد حافظه (RAM) شود.

### 5.1 خواندن فایل خط به خط برای جلوگیری از مصرف زیاد حافظه ✓

به‌جای:

```
with open("bigfile.txt", "r") as file:  
    data = file.read() # ممکن است حافظه زیادی مصرف کند
```

از:

```
with open("bigfile.txt", "r") as file:
    for line in file:
        print(line.strip()) # خواندن و پردازش خط به خط
```

✓ مزیت:

- مصرف حافظه کاهش می‌یابد.
- مناسب برای پردازش فایل‌های بزرگ.

## ✓ 5.2 نوشتن فایل به صورت بافر (buffered)

اگر قصد نوشتن حجم زیادی از داده را دارید، بهتر است از حالت بافر برای بهینه‌سازی سرعت استفاده کنید.

```
with open("big_output.txt", "w", buffering=8192) as file: # بافر ۸ کیلوبایت
    file.write("مقدار زیادی از داده‌ها")
```

✓ مزیت: افزایش سرعت نوشتن در فایل‌های حجیم.

## 🎯 جمع‌بندی نکات کلیدی

- ✓ از "a" به جای "w" استفاده کنید تا داده‌های قبلی حذف نشوند.
- ✓ برای جلوگیری از خرابی فایل، ابتدا در یک فایل موقت بنویسید و سپس جایگزین کنید.
- ✓ قبل از نوشتن، صحت داده‌ها را بررسی کنید تا اطلاعات نادرست ذخیره نشوند.
- ✓ از قفل فایل (filelock) استفاده کنید تا پردازش‌های موازی باعث تداخل نشوند.
- ✓ مراقب مسیرهای فایل باشید و اجازه ندهید کاربر مسیر را تعیین کند.
- ✓ برای فایل‌های حجیم، از خواندن خط به خط و نوشتن بافر شده استفاده کنید.

## 🎯 تمرین:

یک برنامه بنویسید که:

1. نام و ایمیل کاربران را دریافت کند.
2. بررسی کند که ایمیل معتبر باشد.
3. داده را در فایل موقت ذخیره کند و سپس آن را جایگزین فایل اصلی کند.