Practical 08: Advanced Topics and Image Manipulation

Dr. Todd Jones

t.r.jones@reading.ac.uk

Department of Computer Science

Follow the instructions to complete each of these tasks.

This set of exercises focuses on

This work is not assessed but will help you gain practical experience for the coursework.

The Walrus Operator

Write a function that processes sensor data read from a file. Use the walrus operator to read lines until an empty line is encountered, updating a running total of sensor readings.

Remember:

- The walrus operator (:=) combines reading and testing in one step.
- Typically, the operator should be used only when it improves clarity.
- A complete solution should include **error handling for invalid readings**.

Some starter code has been provided:

In [ ]:
```python
def process_sensor_data(file_path: str):
    total = 0.0
    # TODO: Use the walrus operator to read lines and update total.
    pass

# Assume 'sensor.txt' contains one reading per line.
process_sensor_data("./data/sensor.txt")
```

Decorators

Write a decorator called `cache_and_time` that both caches a function's output (to avoid repeated computation) and prints the execution time every time the function is called. Apply it to a function that simulates an expensive computation (e.g., summing the squares of numbers).

Remember:

- A `functools.wraps` call will preserve the wrapped function's metadata.
- to check the cache **before** measuring time.

Think about how to construct a **reference** to a unique argument.

Note: this is similar to the lecture exercise but asks you to create your own caching functionality.

Some starter code has been provided:

In [ ]:
```
import time
import functools

def cache_and_time(func):
    # TODO: implement caching and timing.
    pass

@cache_and_time
def expensive_computation(n: int) -> int:
    # Simulate an expensive operation
    time.sleep(2)
    return sum([x * x for x in range(n)])

# Test the decorated function
print(expensive_computation(10))
print(expensive_computation(10))
print(expensive_computation(20000))
```

Dataclasses

Create a `dataclass` named `Book` with **keyword-only parameters** for `title`, `author`, and `price`.

Additionally, use a **default factory** for an attribute, `tags`, (a list of strings) to describe subjects that can help to identify the category of book that has been recoreded.

You might enable `slots` to optimise memory usage.

Then, write a function to print a formatted **summary** of a book instance.

Remember:

- The `@dataclass` decorator **automatically** generates initialisation and representation methods.
- Using `kw_only=True` forces keyword-only arguments for clarity (requires 3.10+).
- The default factory, by way of the `field` module, for `tags` ensures each instance gets its own list.
- Enabling `slots` reduces memory usage, a subtle but important optimisation for large numbers of objects (requires 3.10+).
- Type **annotations** are important to configure a `dataclass`.

Some starter code has been provided:

In [ ]:
```
from dataclasses import dataclass, field

# TODO: Define the Book dataclass with kw-only parameters, default factory, and slots.

def print_book_summary(book):
    # TODO: Print a formatted summary of the book.
    pass
```

```python
# Test the dataclass
book1 = Book(title="Python Unleashed", author="Alex Doe", price=39.99)
print_book_summary(book1)
```

Dataclasses Extension

Extend the `Book` exercise by adding the following features:

- Implement a `@classmethod` named `from_csv` that accepts a **comma-separated string** (with tags separated by semicolons) and returns a new `Book` instance.
  - In the domain of book management, data is frequently exchanged in CSV format (e.g., a catalogue export or database dump).
  - This will work as an **alternative constructor** that enables the creation of a `Book` instance without key words.
  - It might help to enforce at least the necessary number of inputs.
- Create a `@staticmethod` named `is_valid_price` that determines whether a given price is sensible (for instance, greater than *zero* and less than 1000).
  - In the book domain, ensuring reasonable price values helps to maintain data integrity, likely supporting business rules, pricing policies, and consumer satisfaction.
  - Making this method independent of any particular instance, it is ideal for validating data.
  - Decoupling means that it can be used in various parts of an application, including separate user input validation or API endpoints.
- Add a `@property` named description that returns a **formatted description** of the book. This description should include the original price and a discounted price. The discount should be computed as a function of the number of `tags`, but shall not exceed 50% of the original price.
  - In the book domain, clear and attractive presentation of information is desirable. It might also be useful for logging.
  - The property provides a computed, **read-only** attribute, incorporating dynamic generation of additional information, reducing the need for explicit function calls to retrieve common information

In [ ]:

```python

```

Logging

In the security domain, timely email notifications can aid in mitigating potential breaches and ensuring system integrity. Or maybe you just want to set up some personal notifications that you can receive while on the move.

Develop a script (can run from notebook) that simulates processing **security events**. For each event, described by an **integer severity value**, log an `INFO` message when processing begins and log an `ERROR` message if the event severity exceeds a defined threshold (for example, greater than 7).

Configure **two logging outputs**:

1. to the console
2. via email using an `SMTPHandler`

The email hanlder is described here:
https://docs.python.org/3.11/library/logging.handlers.html#smtphandler

Actually configuring an email may be challenging based on system/provider/organisation settings. To **SIMULATE** an email to test this process, we can configure a **local debugging SMTP server** via built-in Python tools.

1. In a new console/terminal with Python access, execute: `python -m smtpd -c DebuggingServer -n localhost:1025` .

- This may show warnings but otherwise show **no response**.
- This launches a local SMTP server on port 1025 that does not actually send emails; instead, it prints the email contents to the console.

2. Configure the `SMTPHandler` as follows:

```
smtp_handler = SMTPHandler(
    mailhost=("localhost", 1025),
    fromaddr="your_email@example.com",
    toaddrs=["recipient@example.com"],
    subject="Security Alert - Critical Error",
    credentials=None,  # No credentials are needed for the debugging server.
    secure=None        # No secure connection is required.
)
```

3. Example output below.

Alternatively, if you would like to see a real email reliably be sent, you can register for a free account with a service, like https://mailtrap.io/, configuring an email via API/SMTP. Locating the associated credentials, you can configure the SMTPHandler like:

```
smtp_handler = SMTPHandler(
    mailhost=("live.smtp.mailtrap.io", 587),  # Mailtrap SMTP server and port.
    fromaddr="Private Person <hello@demomailtrap.co>",  # Sender's address.
    toaddrs=["A Test User <xx123456@reading.ac.uk>"],      # Recipient's address as
a list.
    subject="Hi Mailtrap",                                 # Email subject.
    credentials=("api", "2d3d6df5eaf49da50525c476bde2e"),  # SMTP authentication
credentials.
    secure=()  # An empty tuple signals to initiate TLS (equivalent to calling
starttls()).
)
```

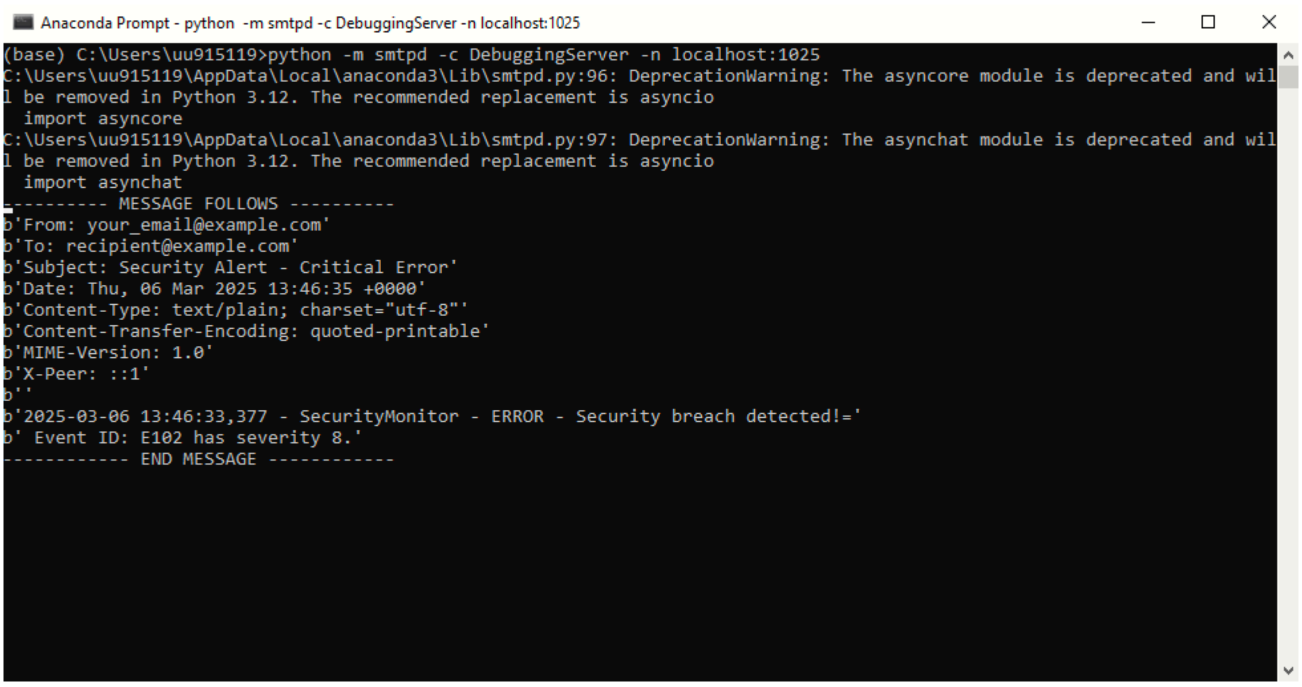
- **Don't forget to check junk/spam box!**

Some starter code is provided:

In [ ]:
```
import logging  # Import the logging module.
from logging.handlers import SMTPHandler  # Import SMTPHandler for email alerts.

# ------------------------------------------------------------------------
# Logger Configuration
# ------------------------------------------------------------------------

# Create a logger for security events.
logger = logging.getLogger("SecurityMonitor")
logger.setLevel(???)

# ------------------------------------------------------------------------
# Console Handler Configuration
```

```python
# ----------------------------------------------------------------------------

# Create a console handler to display log messages on the terminal.
console_handler = 
console_handler.setLevel(???)
# ...

# ----------------------------------------------------------------------------
# SMTP Handler Configuration (for email alerts)
# ----------------------------------------------------------------------------

# Configure an SMTP handler to send email alerts for ERROR level messages.
# Replace the placeholder SMTP details with your own credentials and server information.
smtp_handler = SMTPHandler(
    mailhost=("your_smtp_server", 587),      # e.g., ("live.smtp.mailtrap.io", 587) for Mailt
    fromaddr="sender@example.com",            # Replace with the sender's email address.
    toaddrs=["recipient@example.com"],        # Replace with the recipient's email address(es
    subject="Security Alert",                 # Subject line for the email alert.
    credentials=("your_username", "your_password"),  # Replace with valid SMTP credentials.
    secure=()  # An empty tuple indicates that TLS is to be used (equivalent to starttls()).
)
smtp_handler.setLevel(???)
# ...

# ----------------------------------------------------------------------------
# Function to Process Security Events
# ----------------------------------------------------------------------------

def process_security_event(event):
    """
    Process a security event by logging its start and sending an alert if the event is severe
    """
    # Log an INFO message when processing begins.
    # ...

    # If the event severity exceeds a threshold (greater than 7), log an ERROR message.
    # ERROR messages will trigger an email alert via the SMTP handler.
    # ...

# ----------------------------------------------------------------------------
# Main Execution
# ----------------------------------------------------------------------------

# A sample list of security events.
events = [
    {"id": "E101", "severity": 5},
    {"id": "E102", "severity": 8},
    {"id": "E103", "severity": 3}
]

# Process each security event.
for event in events:
    process_security_event(event)

# ----------------------------------------------------------------------------
# Optional: Clean up logging handlers
# ----------------------------------------------------------------------------
# In a standalone script, logging shutdown occurs automatically upon interpreter termination.
# For long-running applications, it may be appropriate to remove and close handlers explicitl

logger.removeHandler(smtp_handler)
smtp_handler.close()
logger.removeHandler(console_handler)
console_handler.close()
logging.shutdown()
```

Console generated email:


Anaconda Prompt - python -m smtpd -c DebuggingServer -n localhost:1025

```
(base) C:\Users\uu915119>python -m smtpd -c DebuggingServer -n localhost:1025
C:\Users\uu915119\AppData\Local\anaconda3\Lib\smtpd.py:96: DeprecationWarning: The asyncore module is deprecated and wil
l be removed in Python 3.12. The recommended replacement is asyncio
  import asyncore
C:\Users\uu915119\AppData\Local\anaconda3\Lib\smtpd.py:97: DeprecationWarning: The asynchat module is deprecated and wil
l be removed in Python 3.12. The recommended replacement is asyncio
  import asynchat
---------- MESSAGE FOLLOWS ----------
b'From: your_email@example.com'
b'To: recipient@example.com'
b'Subject: Security Alert - Critical Error'
b'Date: Thu, 06 Mar 2025 13:46:35 +0000'
b'Content-Type: text/plain; charset="utf-8"'
b'Content-Transfer-Encoding: quoted-printable'
b'MIME-Version: 1.0'
b'X-Peer: ::1'
b''
b'2025-03-06 13:46:33,377 - SecurityMonitor - ERROR - Security breach detected!='
b' Event ID: E102 has severity 8.'
------------ END MESSAGE ------------
```

Pillow from NumPy

Generate a checkerboard pattern image using the Pillow library and numpy.

1. Create a numpy array where each element represents one board square (8x8).
2. The pattern shall alternate between two colours, **purple and orange**.
3. When the image is displayed, each square should remain clearly defined regardless of the image size.

Hints:

- Use numpy's `indices` function to create coordinate arrays and compute an alternating pattern using the modulo operator.
- Convert the numpy array to a Pillow Image object using `Image.fromarray`.
- Use the resize method with the NEAREST resampling filter to ensure that the checkerboard squares remain distinct even when the image is enlarged.

```
In [ ]:
```

Adding playing pieces

Extend the functionality to place **circular playing pieces** on the top and bottom three rows of the board, but only on the squares that are designated for play (i.e., the dark squares, as in draughts).

Ensure that the pieces are drawn centred within the square, with a slight margin.

Hint:

- It will help to save and operate on the earlier pattern (binary array).
- Opposing player pieces should have different colours.

In [ ]:

Green Screen

In the `./data/` directory, you will find a series of images. One of them, `todd.jpg` is a picture of your lecturer in front of a green board.

The rest of the images depict holiday destinations, sourced from https://commons.wikimedia.org.

Send Todd on holiday:

1. Using pillow, choose a holiday destination image to use as a background.
2. Replace the green board pixel's in Todd's image with transparent ones.
3. `paste` the resulting image on the background.

Note:

- The images have varying dimensions; resizing may be needed.
- The green pixels are not very uniform. Some thresholding is required to identify pixels of a similar colour, and multiple passes might be needed.
- Numpy is an efficient way to quickly process all pixels.

In [ ]:

Adding text (optional)

Add text to the image you have created, styling it like a postcard.

In [ ]: