Advanced Topics and Image Manipulation

Dr. Todd Jones

t.r.jones@reading.ac.uk

Department of Computer Science

Introduction

Advanced Topics and Image Manipulation

Today we will explore the the capabilities of the `Pillow` library--the modern fork of `PIL`--and gain familiarity with a selection of **advanced Python techniques** that can can further enhance your knowledge of Python resources to make your code more **concise**, **modular**, and **efficient**.

Learning Objectives

Pillow Fundamentals:

- Learn how to read, display, and write images, and perform basic transformations (resizing, cropping, rotating, blending, masking, and annotating).

Advanced Python Concepts in Action:

- The **walrus operator** will show you how to simplify loops and conditional assignments, cutting down on repetitive code.
- **Closures** will introduce you to the concept of functions that remember their context, a cornerstone for building reusable components.
- With **decorators**, you'll see how to elegantly add cross-cutting features (like logging and caching) to your functions without cluttering their logic.
- **Dataclasses** demonstrate how you can write cleaner, self-documenting code for data models, enabling you to focus on business logic rather than boilerplate.

The Walrus Operator

A neat piece of **syntactic sugar** introduced in *Python 3.8*.

`:=` is an **assignment expression** that lets you assign values to variables as part of larger expressions.

- can make your code more concise by combining the computation and assignment in one step

- especially handy in loops, comprehensions, and conditional statements

Traditionally, you would need to compute a value on one line and then use it in a condition on the next. With the walrus operator, you can both compute the value and assign it to a variable within the same expression.

The operator looks like `:=` (its shape often reminds people of a walrus, hence the nickname). It takes the value on its right-hand side, **assigns** it to the variable on its left-hand side, and then **returns** that value.

Using the Walrus Operator in a `while` Loop

When reading input until a blank line is entered, you can use the walrus operator to both read and test the input in one line:

In [ ]:
```
'''
Here, line := input(...) reads user input and assigns it to line at the same time.
The loop continues until an empty string is entered.
'''

while (line := input("Enter something (or leave blank to quit): ")) != "":
    print(f"You entered: {line}")  # Access the newly assigned variable.

print("Goodbye!")
```

Simplifying Conditional Checks

You can use the walrus operator in an `if` statement to capture a value and then use it immediately:

In [ ]:
```
'''
Instead of calling len(nums) twice, the walrus operator assigns the length
  to n and simultaneously checks if it is greater than 3.
At the same time, we assign and return 3 to thresh.
'''

nums = [1, 2, 3, 4, 5]

if (n := len(nums)) > (thresh := 3):
    print(f"The list has {n=} elements, which is more than {thresh=}.")
```

Combining Assignment in a List Comprehension

The walrus operator can also streamline **list comprehensions** when you need to both **compute and filter** based on a value:

In [ ]:
```
'''
For each number n in nums, the expression (square := n * n)
  computes and assigns its square to square.
The comprehension then includes only those square values that exceed 10.
'''

nums = [1, 2, 3, 4, 5]
```

```python
# Compute the square and filter only those squares that are greater than 10.
squares = [square for n in nums if (square := n * n) > 10]

print(squares)  # Output: [16, 25]
```

With out the walrus operator, the above operation cannot be performed with a comprehension. Instead, we would need to rely on a loop, computing the square once per iteration:

```python
result = []
for n in nums:
    square = n * n
    if square > 10:
        result.append(square)
```

Notes on Walrus Operator Utility

Conciseness and Readability

- By reducing repetition (e.g., avoiding multiple calls to the same function), the walrus operator can simplify code and reduce the risk of mistakes.

Performance

- Where computing a value is expensive (such as reading from a file or making a network request), using the walrus operator prevents the need for redundant computations.

Use Cases

- *Repeated overwriting assignments with reuse.*
- Loops: Read and process data until a termination condition is met.
- Conditionals: Assign a value while testing it in if statements.
- Comprehensions: Avoid repeating calculations inside filters.

Caution

- Overusing the walrus operator or using it in very complex expressions can sometimes **harm readability** for others. It is best used when it genuinely reduces redundancy and clarifies the code.
- Sometimes, it can be just as effective, if not clearer, to simply store a value in a variable via common assignment, particularly if the value is unlikely to change frequently.

Closures

A **closure** is essentially a nested function that "remembers" the state (the local variables) of its **enclosing scope** even after that scope has completed.

- a way to attach data (state) to a function without using global variables or classes.

When you define a function inside another function, the inner function has access to the outer function’s variables.

When you return the inner function, it carries with it a reference to those variables. This is the essence of a closure.

In this example, the outer function `make_multiplier` takes a `factor` and returns a new function that multiplies its input by that `factor`.

The inner function, `multiplier,` remembers the value of `factor` even after `make_multiplier` has returned.

This allows you to create specialized functions on the fly.

In [ ]:
```
def make_multiplier(factor):
    def multiplier(n):
        # 'factor' is captured from the enclosing scope
        return n * factor
    return multiplier

# Create a closure that multiplies by 3
times3 = make_multiplier(3)
print(times3(5))  # times3 remembers the 3 factor

# Create another closure that multiplies by 10
times10 = make_multiplier(10)
print(times10(5))  # times10 remembers the 10 factor
```

This example uses a closure to **maintain state**.

The `make_counter` function returns a function that increments and returns an **internal count**.

Even though `make_counter` finishes executing after returning `counter`, the inner function still has access to the variable `count` from its defining environment.

This is accomplished through the `nonlocal` keyword.

- See: https://www.w3schools.com/python/ref_keyword_nonlocal.asp

In [ ]:
```
def make_counter():
    count = 0
    def counter():
        nonlocal count  # Allows modifying the variable 'count' from the enclosing scope
        count += 1
        return count
    return counter

# Create a counter and call it multiple times
my_counter = make_counter()
print(my_counter())  # Output: 1
print(my_counter())  # Output: 2
print(my_counter())  # Output: 3
```

Recall: We have encountered this functionality before, in somewhat more complex form:

In [ ]:
```
def power(f, n):
    """
    Create a function that calls itself n times
    """

    def inner(x):
        """
        Constructor function to apply f n times using input x
        """
        # Iterate the call of f n times,
        # reusing the previous result as new input each time
        for _ in range(n):
            x = f(x)
        return x

    return inner

# Create g from f and n, passed to power
n = 3
f = lambda x: x**2
g = power(f, n)


# Test g with x=2 and x=3, comparing to expected solution
x = 2
print(f'{x=}, {g(x)=}, explicit: {((x**2)**2)**2}')
x = 3
print(f'{x=}, {g(x)=}, explicit: {((x**2)**2)**2}')
```

Notes about the example solution, for the `power` function:

In the above, we define a function `power(f, n)` that takes two arguments: a function `f`, and an integer `n`. The function `power()` returns another function `inner()` that takes one argument `x`. This result from `power(f, n)` is `g`.

The inner function `inner(x)` uses a `for` loop to apply *specifically* the function `f` recursively `n` times based on some initial input, `x`. We update `x` in each iteration for subsequent use as input.

The argument to the resulting function, `g`, is the first `x` that will be used in the nested function.

With the example values and function used, $f(x) = x^2$, and with $n = 3$, $g(x) = f(f(f(x))) = x^{2^{2^2}}$. For $x = 2$, this is processed iteratively in 3 steps as:

- $2^2 = 4$
- $4^2 = 16$
- $16^2 = 256$

Notes on the Utility of Closures

Encapsulation Without Classes

- Closures provide a lightweight way to bundle functionality with data. They are especially useful when you want to **avoid creating a full class** just to maintain some state.

Functional Programming

- In functional programming, functions are **first-class** "citizens". Closures are a natural extension, letting you create function factories (functions that produce specialized functions).

Data Hiding

- Closures help you hide data. Variables captured in a closure are **not accessible** from the global scope, reducing the risk of unintended modifications.

Building Decorators

- Many decorators (coming up next) are implemented as closures. These capture the original function and add extra behavior around it.

Improving Code Readability

- By packaging related functionality and its context together, closures *can* lead to cleaner, more modular code.

NOTE:

- Deeply nested closures can be hard to follow.
- Variables captured by closures are read-only unless you use mutable objects or the `nonlocal` keyword carefully.
- Closures may inadvertently hold onto large objects if not managed correctly.

Choosing Closures vs. Classes

Aspect	Closures	Classes
Simplicity	Concise and lightweight for small tasks	More boilerplate but clearer structure for complex systems
State Management	Captures limited state from the enclosing scope	Designed for **managing** more complex state with multiple attributes/methods
Behavior	Ideal for **single-purpose** functionality	Better for modeling objects and complex behaviors, including **inheritance**
Overhead	Minimal overhead; no need for a separate structure	Additional structure may be required, offering features like polymorphism
Use Case	Use when you need a **temporary** function capturing some state	Use when building **scalable, organized** code with multiple interrelated methods

---

Decorators

Building on our exploration of closures, we now turn to **decorators**, a powerful tool that leverages closures to modify or extend the behaviour of **functions and classes** without altering their core logic.

Decorators are essentially *syntactic sugar* for **wrapping** functions or classes, allowing you to **inject additional functionality** (like logging, caching, performance monitoring, etc.) in a clean, reusable way.

More specifically, decorators are **higher-order functions** that

- take another function or class as an argument
- returns a new function or class that usually implements/calls the original with some extra behaviour.

Syntax:

```
@decorator
def function(args):
    statements...
```

We will rely on the `functools` library, which supplies utilities for manipulating higher-order functions.

- Mainly, the `@functools.wraps(funcion)` decorator...to make...decorators...
- See: https://docs.python.org/3.11/library/functools.html

But why use decorators?

Separation of Concerns

- They let you separate auxiliary tasks (e.g., logging, caching, input validation) from core business logic.

Reusability

- Once written, a decorator can be applied to many functions or classes to add the same behaviour consistently.

Readability and Maintainability

- By isolating repetitive code in decorators, primary functions and classes remain focused on their main purpose.

Where You Will Find Them

- Decorators are used extensively in **web frameworks** (e.g., for routing, authentication), data processing (e.g., for caching API responses), and even in system-level programming (e.g., for performance monitoring).

NOTE:

- Deeply **stacked** or overly clever decorators can obscure the control flow, making debugging harder.
- The **order** in which multiple decorators are applied matters and can lead to unexpected behaviour.

Examples

Scikit-learn offers a `deprecated` decorator to mark functions as outdated.

- When such a wrapped/modified function is called, it emits a warning to inform users that the function will be removed in future versions.

When running the following code, we will receive a deprecation warning indicating that `old_function` is outdated.

In [ ]:
```python
from sklearn.utils.deprecation import deprecated

@deprecated("This function is deprecated and will be removed in a future version.")
def old_function(x):
    return x * 2

print(old_function(5))
```

A very commonly used decorator is the built-in `@lru_cache` from the `functools` module.

- LRU: **L**east **R**ecently **U**sed items are discarded first, https://docs.python.org/3.11/library/functools.html#functools.lru_cache
- **Caches** the results of function calls.
- If the same inputs occur again, Python can **return the cached result immediately** rather than recomputing it.
- This can significantly speed up functions that are **expensive** to compute or are **called repeatedly** with the same arguments.

In [ ]:
```python
from functools import lru_cache

# maxsize=None means equivalent to @cache from functools
# Otherwise, we limit the size of the cache

@lru_cache(maxsize=None)
def expensive_computation(x):
    # Simulate an expensive computation
    print(f"Computing for {x}...")
    return x * x

# First call computes the result and caches it.
print(expensive_computation(10), '...slowly')

# Second call retrieves the result from the cache without recomputation.
print(expensive_computation(10), '...quickly')
```

This is **easy to apply** and can **drastically reduce runtime** for recursive or repeatedly-called functions.

A Custom Timer Decorator

This decorator measures and prints the execution time of a function. It is useful for performance testing or profiling code.

In [ ]:
```python
import time
from functools import wraps

def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()              # Record the start time
        result = func(*args, **kwargs)        # Call the original function
        end_time = time.time()                # Record the end time
        print(f"{func.__name__} executed in {(end_time - start_time) * 1e9:,.4f} ns")
        return result
    return wrapper

# Usage of the timer decorator
@timer
def example_func():
    time.sleep(1)  # Simulate a delay
    return "Done!"

print(example_func())
```

Now, we can extend and wrap our cached function in the `timer` decorator.

- That's right, double wrapping!

```
In [ ]: from functools import lru_cache

        # maxsize=None means equivalent to @cache from functools
        # Otherwise, we limit the size of the cache

        @timer
        @lru_cache(maxsize=None)
        def expensive_computation(n):
            # Simulate an expensive computation
            print(f"Computing for {n}...")
            time.sleep(1)
            if n < 2:
                return 1
            return n * expensive_computation(n - 1)

        # First call computes the result and caches it.
        print(expensive_computation(10))

        # Second call retrieves the result from the cache without recomputation.
        print(expensive_computation(10))
```

```
In [ ]: print(expensive_computation(11))
```

Further exploration: What happens if you reverse the order of the wrapping???

The `timer` decorator is applied first, and then `lru_cache` wraps the result of that decorated function. On the first call, the function runs and its execution time is measured by timer, then the result (including the fact that it has been "timed") is cached by `lru_cache`. On subsequent calls, `lru_cache` returns the cached result directly without calling the inner timer wrapper. As a result, you don't see any `timer` reporting on the cached call because the timing code isn’t executed again.

Building a Logging Decorator

Let's go back to the construction of a decorator.

Imagine we are developing a billing system and want to log (i.e., write a status message) every time a discount is calculated.

Here, the we create the `log_call` decorator to **wrap** the `calculate_discount` function, printing its inputs and output.

This pattern is useful in production systems where tracking function behaviour is needed for debugging and auditing.

```
In [ ]: import functools

        def log_call(func):
            """A decorator that logs the function call details."""
            ???


        @log_call   # Try this cell again with this line commented out.
        def calculate_discount(amount, discount_rate):
            """Calculate discounted price for a purchase."""
            return amount * (1 - discount_rate)
```

```
# Example usage in a billing system
final_price = calculate_discount(250, 0.15)
print(f"Final price: {final_price}")
```

Decorators for Classes

Decorators are not limited to functions; they are equally powerful when applied to classes or their methods.

You can use class decorators to:

- **Modify or Augment Class Behaviour:**
  - Automatically add or modify methods and attributes.
- **Implement Design Patterns:**
  - Such as creating singletons or automatically registering classes.
- **Simplify Class Definitions:**
  - For example, the `@dataclass` decorator in Python automatically generates special methods like `__init__` and `__repr__` for classes that primarily store data.

Additionally, Python includes built-in decorators like:

- `@classmethod` and `@staticmethod` to define methods that behave differently from regular instance methods.
- `@property` to create managed attributes with getter, setter, and deleter functionalities.

Modify or Augment Class Behaviour

A class decorator can modify the class object itself, letting us automatically add or modify methods and attributes at the time the class is defined.

This can be useful when enhancing classes with common functionality, such as logging, validation, or custom behaviors.

An implementation example:

In [ ]:
```
def add_methods(cls):
    # Automatically add a new attribute to the class
    cls.new_attribute = "I am an automatically added attribute."

    # Automatically add a new method to the class
    def new_method(self):
        return f"New method called! The original value is: {self.value}"

    cls.new_method = new_method
    return cls

@add_methods
class Example:
    def __init__(self, value):
        self.value = value

# Creating an instance of Example
e = Example(42)

# The instance now has the additional attribute and method
```

```
print(e.new_attribute)
print(e.new_method())
```

Implementing Design Patterns

Using a decorator to register a class means that when you define the class, the decorator automatically adds the class to a registry or performs some kind of bookkeeping.

This is common in plugin systems or frameworks where you want to keep track of all available implementations without **manually maintaining** a list.

For example, a simple registry decorator might look like this:

In [ ]:
```
# A global registry to hold registered classes
registry = {}

def register_class(cls):
    """Decorator that registers a class by its name."""
    registry[cls.__name__] = cls
    return cls

@register_class
class MyComponent:
    pass

@register_class
class AnotherComponent:
    pass

# Inspect registry result
registry
```

Using decorators for **singletons** (classes for which there should be only one instance in cases of tight control or management of resources or configurations) applies a similar concept but focuses on **controlling the instantiation** of a class so that only one instance ever exists.

A singleton decorator intercepts calls to create a new instance and returns the same instance if one has already been created.

For example, perhaps we desire management of a single connection to a database ensuring efficient resource use and consistent transaction management.

- The first time a `DatabaseConnection` is instantiated, the decorator creates and caches an instance.
- Subsequent calls simply **return the cached instance**, ensuring that only one instance exists regardless of how many times you "create" it.

In [ ]:
```
import functools

def singleton(cls):
    """A decorator that transforms a class into a singleton by caching its instance."""
    instances = {}

    @functools.wraps(cls)
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class DatabaseConnection:
    def __init__(self, connection_string):
        self.connection_string = connection_string

# Both a and b refer to the same instance.
a = DatabaseConnection("db://localhost")
b = DatabaseConnection("db://remote")
print('Are the two objects the same?  ', a is b)
print('Inspect database connection:  ', b.connection_string)
```

Dataclasses

Dataclasses simplify the creation of classes meant primarily to store data by automatically generating common methods like `__init__`, `__repr__`, and `__eq__`. They are especially useful when defining lightweight data containers with **minimal boilerplate**. Modern enhancements such as keyword-only parameters, slots, and default factories further empower dataclasses, making them both expressive and efficient.

A `@dataclass` is a class decorator introduced in Python 3.7 that automatically adds special methods to user-defined classes. Adding `kw_only=True`, `slots=True`, and using `field(default_factory=...)`, we can fine-tune how your data container behaves:

- `kw_only=True`: Ensures that certain fields can only be provided as keyword arguments, enhancing clarity and reducing errors.
- `slots=True`: Generates a `__slots__` attribute for memory efficiency and faster attribute access by preventing the creation of a per-instance `__dict__`.
- `default_factory`: Provides a way to set dynamic default values (such as an empty list) for fields.
    - Prevents shared state across instances.

See: https://docs.python.org/3.11/library/dataclasses.html

Syntax: To construct a dataclass, we need to **annotate class attributes with types**.

- The `@dataclass` decorator uses these annotations to create a constructor and other methods automatically.
- Generally, Python will require type annotations to construct a dataclass; the whole point of a dataclass is to declare fields with types so that Python knows what data to expect.
    - This is also useful for static **type checking**.

Here, `name` and `price` are automatically made into fields of the class:

In [ ]:
```
from dataclasses import dataclass

@dataclass
class Product:
    ???
```

In [ ]:
```
# The type annotations are stored as a dictionary.
Product.__annotations__
```

In [ ]:
```python
# Inspect the initialisation signature
?Product
```

In [ ]:
```python
# Attempt instantiation, inspect error details
p = Product()
```

Let's look at a more complex example.

Imagine we are building a billing system where each **order** holds:

- a customer name
- an order ID
- a list of purchased items
- a running total for the complete order.

Using a `dataclass` makes it easy to define this **container**:

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

@dataclass(kw_only=True, slots=True)
class Order:
    order_id: int
    customer: str
    items: list[str] = field(default_factory=list)
    total: float = 0.0

    def add_item(self, item: str, price: float):
        self.items.append(item)
        self.total += price

# Creating an order using keyword-only arguments
order = Order(order_id=101, customer="Alice")
order.add_item("Wireless Mouse", 25.99)
order.add_item("Keyboard", 45.50)
print(order)
```

The `dataclass` automatically creates an initialisation routine that requires `order_id` and `customer` to be provided as **keywords**.

The `items` list is initialised via a `default_factory`, ensuring each order gets its own **list**, and `slots=True` is used to improve memory efficiency.

In this case, the `__repr__` is automatically constructed and used via the `__str__`.

Another example:

In an IoT context, capturing sensor data along with a **timestamp** and a series of measurements may be important. A `dataclass` can serve as a compact container for this purpose.

In the following scenario, `SensorData` automatically tracks when data is captured via a **default timestamp**.

The `readings` field uses a `default_factory` to ensure a new list is created for each instance. This pattern is especially useful in real-world applications where encapsulating state is crucial.

Note: In this example, we use the pre-Python 3.9 version of **type hinting**, which requires that we import the `List` from the `typing` module, as opposed to the more recent built-in generic type hinting via `list`.

In [ ]:
```python
from dataclasses import dataclass, field
from datetime import datetime
from typing import List

@dataclass(kw_only=True, slots=True)
class SensorData:
    sensor_id: str
    readings: List[float] = field(default_factory=list)
    timestamp: datetime = field(default_factory=datetime.now)

    def add_reading(self, value: float):
        self.readings.append(value)

    def average(self) -> float:
        return sum(self.readings) / len(self.readings) if self.readings else 0.0

# Simulate collecting sensor data
data = SensorData(sensor_id="temp_sensor_007")
data.add_reading(22.5)
data.add_reading(23.0)
data.add_reading(22.8)
print(data)
print("Average Temperature:", data.average())
```

CAUTION: Type hints are just that: **hints**.

To enforce static typing, explore tools like:

- `mypy` - A static type checking utility to analyse written code, similar to how a compiler might check types.
  - https://mypy-lang.org
- `Typeguard` - A library that facilitates (via decorators) inspection of type hints, raising errors for nonconforming values at runtime.
  - https://github.com/agronholm/typeguard
- `pydantic` - A framework to enforce (via class inheritance) data validation when creating objects, potentially converting types.
  - https://docs.pydantic.dev

In [ ]:
```python
# Create instance with nonconforming types
data = SensorData(sensor_id=6, readings=['1', '2'])
print(data)
```

In [ ]:
```python
print("Average Temperature:", data.average())
```

Classmethod

A `classmethod` is a method that receives the class itself as the first argument (usually named `cls`) rather than an *instance* of the class.

This decorator is useful for factory methods or methods that need to affect the class state.

- E.g.: define a method to initialise a class from a non-standard data type by parsing a string instead of some expected string and integer arguments.

In [ ]:
```python
class MyClass:
    @classmethod
    def create(cls, value):
        cls.state = value
        return cls()
```

Staticmethod

A `staticmethod` is essentially a function defined inside a class that **does not** take the implicit first parameter (`self` or `cls`).

It behaves like a *plain function* that happens to reside in the class's namespace.

- E.g.: A `Circle` class might have an `area` `staticmethod` that does not rely on instance-specific data, providing a utility for external data (and still potentially used internally).

In [ ]:
```python
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b
```

Method Type	Definition	Implicit First Parameter	Common Use Cases	Key Characteristics
Standard Method	A regular instance method	`self`	Manipulating or accessing instance-specific data and behavior	Must be called on an instance; can access and modify instance attributes; bound to a specific object.
Class Method	A method that operates on the class itself	`cls`	Factory methods, alternative constructors, modifying class state	Decorated with `@classmethod` ; can be called on both the class and its instances; cannot access instance data.
Static Method	A function within a class's namespace that does not depend on class or instance data	None	Utility functions that logically belong to the class	Decorated with `@staticmethod` ; can be called on the class or an instance; no access to `cls` or `self` .

Property

The `property` decorator allows you to define methods in a class that can be accessed like **attributes**.

This is useful for computed properties or for **controlling access** (getters, setters, and deleters).

Usage:

- You define a method with the `@property` decorator, and optionally define setters and deleters with `@<propertyname>.setter` and `@<propertyname>.deleter` .
- See: https://docs.python.org/3/library/functions.html#

In the example below, we define a `Patient` class that manages a patient's heart rate.

- The heart rate is encapsulated as a property with a getter, setter (with validation for a reasonable range), and a deleter.
- We also include a computed property that indicates whether the patient has tachycardia (an abnormally high heart rate).

In [ ]:
```python
class Patient:
    def __init__(self, name: str, heart_rate: int):
        self.name = name
        # Initialize the internal heart rate attribute.
        self._heart_rate = heart_rate

    # Getter: Access the current heart rate.
    @property
    def heart_rate(self) -> int:
        print(f"Accessing {self.name}'s heart rate")
        return self._heart_rate

    # Setter: Update the heart rate with validation.
    @heart_rate.setter
    def heart_rate(self, value: int) -> None:
        print(f"Updating {self.name}'s heart rate")
        if not (30 <= value <= 200):
            raise ValueError("Heart rate must be between 30 and 200 BPM.")
        self._heart_rate = value

    # Deleter: Delete the heart rate record.
    @heart_rate.deleter
    def heart_rate(self) -> None:
        print(f"Deleting {self.name}'s heart rate record")
        del self._heart_rate

    # Computed property: Check for tachycardia (heart rate > 100 BPM).
    @property
    def tachycardia(self) -> bool:
        print(f"Evaluating tachycardia status for {self.name}")
        return self.heart_rate > 100

# ??? Usage
```

Logging

It is often important to know how to **monitor** and **understand** what happens to that data as it flows through our application.

This is where logging comes in.

Logging provides a systematic way to record runtime events, errors, and other important information.

- Instead of relying on *scattered print statements*, a logging framework helps you capture, store, and later analyse details of your program's execution, which is useful for debugging and production monitoring.
- See: https://docs.python.org/3.11/library/logging.html#logging-basic-tutorial

Python's built-in logging module supports various log levels (`DEBUG` , `INFO` , `WARNING` , `ERROR` , `CRITICAL`) and allows you to configure different handlers (to console, file, etc.) to direct output appropriately. This provides fine-grained control over what information is recorded and where it is stored.

Level	Numeric value	What it means / When to use it
logging.NOTSET	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to NOTSET, then all events are logged. When set on a handler, all events are handled.
logging.DEBUG	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
logging.INFO	20	Confirmation that things are working as expected.
logging.WARNING	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
logging.ERROR	40	Due to a more serious problem, the software has not been able to perform some function.
logging.CRITICAL	50	A serious error, indicating that the program itself may be unable to continue running.

In [ ]:
```python
# From the basic tutorial example:

import logging
logger = logging.getLogger(__name__)
logging.basicConfig(filename='./data/example.log', encoding='utf-8', level=logging.DEBUG)
logger.debug('This message should go to the log file')
logger.info('So should this')
logger.warning('And this, too')
logger.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

In [ ]:
```python
# A more complex implementation, directing output to multiple locations.

import logging
from logging.handlers import RotatingFileHandler
from dataclasses import dataclass, field

# Configure logger
logger = logging.getLogger("OrderLogger")
logger.setLevel(logging.DEBUG)  # Capture all levels of messages

# File handler: log INFO and above to a rotating file
file_handler = RotatingFileHandler("./data/order.log", maxBytes=200, backupCount=5)
file_handler.setLevel(logging.INFO)

# Console handler: log only WARNING and above to the console
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)

# Formatter for consistent log message formatting
# Using LogRecord attributes (see documentation for additional fields)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
console_handler.setFormatter(formatter)

# Attach handlers to the logger
logger.addHandler(file_handler)
logger.addHandler(console_handler)

# Dataclass representing an order
```

```
@dataclass
class Order:
    order_id: int
    customer: str
    items: list[str] = field(default_factory=list)
    total: float = 0.0

    def add_item(self, item: str, price: float):
        logger.debug(f"Order {self.order_id}: Adding item '{item}' at ${price:.2f}.")
        self.items.append(item)
        self.total += price
        logger.info(f"Order {self.order_id}: New total is ${self.total:.2f} after adding '{it

def process_order(order: Order):
    logger.info(f"Processing order {order.order_id} for customer '{order.customer}'.")
    if order.total > 500:
        logger.warning(f"Order {order.order_id}: Total ${order.total:.2f} exceeds $500 limit.
    else:
        logger.debug(f"Order {order.order_id}: Total ${order.total:.2f} is within acceptable

# Simulate order processing in a real-world scenario
order = Order(order_id=101, customer="Alice")
order.add_item("Laptop", 450)
order.add_item("Mouse", 25)
process_order(order)

# Adding another expensive item to trigger a warning
order.add_item("Monitor", 200)
process_order(order)
```

The above will create 4 logging files.

If we shutdown the logger and restart, a new set will be generated and some of the old files will be rotated.

In [ ]:
```
# Explicitly shutdown the logging utility to close active handlers
# In a typical script this is automatic upon normal interpreter termination.
logger.removeHandler(file_handler)
file_handler.close()
logger.removeHandler(console_handler)
console_handler.close()
logging.shutdown()
```

Notes on the Utility of Logging

Diagnostics and Debugging

- Logging provides a historical record of events and errors.
- In production, it is very helpful for diagnosing issues after the fact.

Monitoring Application Health

- By logging critical events, you can monitor the behaviour of your application in real time, which is often essential for performance tuning and ensuring reliability.

Auditing and Compliance

- For systems dealing with financial data, user actions, or other sensitive operations, logs serve as an audit trail.

Seamless Integration with Data Containers

- Just as `dataclasse` s simplify data management, logging helps you understand and track how that data changes over time.
- Together, they can create a robust framework for developing maintainable, real-world applications.

NOTE:

- Excessive logging can **degrade performance** and fill up log files quickly.
- Setting incorrect log levels on handlers might **hide** critical messages or **flood** the output with trivial ones.
- Logging sensitive information (passwords, personal data) can create **security risks**.

Advanced Topics Summary

Today we have explored:

The Walrus Operator

- Combines assignment with evaluation in a single expression, reducing code redundancy.
- Can simplify loops and conditional checks by assigning values on the fly.

Closures

- Functions defined inside other functions that capture and remember their enclosing state.
- Useful for creating function factories and encapsulating behavior without relying on global variables.

Decorators

- Leverage closures to wrap functions and add additional behaviour.
- Versatile mechanism to enhance and maintain code for functinos and classes
- Encapsulate recurring patterns or behaviours in a reusable manner

Logging

- Provides a robust framework for recording runtime events, errors, and performance metrics.
- Essential for debugging, monitoring production systems, and ensuring long-term code quality.

Image Manipulation: A look at Pillow and OpenCV

Pillow is the modern fork of the Python Imaging Library (`PIL`), toolkit for opening, manipulating, and saving various image file formats. It is widely used in web applications, data processing pipelines, and desktop software.

- The following explores the library as a whole, along with practical examples on how to read, display, write, transform, blend, mask, and annotate images.
- See tutorial: https://pillow.readthedocs.io/en/stable/handbook/tutorial.html

Automated image processing is commonly used for tasks like:

- Image resizing
- Format conversion and compression
- Watermarking and copyright protection
- Animated GIF creation
- Response to user specification (GUI)
- Augmenting image datasets for machine learning
- Preprocessing for analysis (greyscale, thresholding)

Introduction to Pillow

The Pillow API provides a mechanism to work with images.

In only a few lines of code, an image may be opened, transformations applied, images blended, and text or shapes added.

Multiple formats are supported, including JPEG, PNG, GIF, BMP, and more.

- **Reading/Writing**: Opening and saving is very easy.
- **Transformations**: Images may be resized, cropped, rotated, and flipped.
- **Blending/Masking**: Images may be combined and masks used for selective operations.
- **Annotating/Drawing**: Text, shapes, or other graphics may be added.
- **Optional Enhancements**: Filters, image enhancements, and additional effects can be utilised.

Reading, Displaying, and Writing Images

Let's begin by opening an image file, displaying it (via the default image viewer), and writing it to a new file.

- Pillow is designed to be efficient with memory.
- When an image is opened, data is often loaded **lazily**, meaning that the full image is not immediately loaded into memory until the pixel data is accessed.
- When saving images, the file extension or an explicit format parameter determines the file type.
  - Different formats have distinct characteristics (e.g., lossy versus lossless compression).

In [ ]:
```python
# Access Pillow
from PIL import Image

# Reading an image
image = Image.open("./data/example.jpg")  # Ensure that 'example.jpg' exists in the working d

# Displaying the image (this opens the default system image viewer, e.g. Photos)
image.show()

# Writing (saving) the image to a new file
image.save("./data/example_copy.png")
```

Some Notes:

- The file extension in the save method determines the output format.

- The image mode (e.g., RGB, RGBA, CMYK, , HSV, L, 1) must be considered; conversion may be required for certain operations.
  - See: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes
  - Some file formats have mode restrictions. JPEG files, for example, do not support an alpha channel. If you try to save an RGBA image as a JPEG, you must convert it to RGB first:

```
if image.mode == "RGBA":
    image = img.convert("RGB")
```

In [ ]:
```
# Understanding the image attributes

print(image.size, image.mode, image.width )
image.split() # Returns individual channel information as Image objects
```

Notebook Functionality

For interactive notebooks, images can be displayed inline using IPython's display utilities:

In [ ]:
```
from IPython.display import display

# Inline display in Jupyter Notebooks
display(image)
```

Basic Transformations

Common transformations such as resizing, cropping, and rotating can be applied via methods applied to the image object.

Note the coordinate system used: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#coordinate-system

- Cartesian pixel coordinate system, with (0, 0) in the **upper left corner**
  - Coordinates refer to the **implied pixel corners**; the centre of a **pixel** addressed as (0, 0) actually lies at (0.5, 0.5).

We will look at: `resize` , `crop` , `rotate` . See also: `transpose` , `thumbnail` , and `ImageOps` , which is useful for working within specific dimensional bounds and creating thumbnails.

For resampling choices and characteristics, see: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#filters-comparison-table

In [ ]:
```
# These methods return a new object


# Resize the image to 50% of its original size
resized_image = image.resize((image.width // 2, image.height // 2))
resized_image.save("./data/resized_example.jpg")

# Crop the image (left, upper, right, lower)
cropped_image = image.crop((100, 100, 400, 400))
cropped_image.save("./data/cropped_example.jpg")
print(cropped_image.size)

# Rotate the image by 45 degrees, retains original dimensions
rotated_image = image.rotate(45)
rotated_image.save("./data/rotated_example.jpg")
```

```
# Rotate the image by 45 degrees (expanded area)
rotated_image = image.rotate(45, expand=True)
rotated_image.save("./data/rotated_expanded_example.jpg")

# Rotate the image by 45 degrees (expanded area, filled)
rotated_image = image.rotate(45, expand=True, fillcolor='orange')
rotated_image.save("./data/rotated_expanded_filled_example.jpg")

# Rotate the image by 45 degrees (expanded area, filled, resampled)
rotated_image = image.rotate(45, expand=True, fillcolor='orange',
                             resample=Image.BICUBIC)
rotated_image.save("./data/rotated_expanded_filled_resampled_example.jpg")
```

Some Notes:

- Resizing without maintaining the aspect ratio can distort images.
- The rotate method creates a new image; if preservation of size is required, the `expand` parameter should be considered.
- Consider how resampling is applied:

In [ ]:
```
centre = [x // 2 for x in image.size]
square = 10
display(image.crop((centre[0] - square, centre[1] - square,
            centre[0] + square, centre[1] + square)).resize((500, 500), resample=Image.NEARES
```

In [ ]:
```
display(image.crop((centre[0] - square, centre[1] - square,
            centre[0] + square, centre[1] + square))
            .resize((500, 500), resample=Image.LANCZOS))
```

Mode Conversion

Use `convert` to change mode, for example to greyscale (`'L'`, `0-255`).

In [ ]:
```
gs = image.convert('L')
display(gs)
```

In [ ]:
```
# Convert to black/white only
square = 100
bw = image.crop((centre[0] - square, centre[1] - square,
            centre[0] + square, centre[1] + square)).convert('1')
display(bw)
```

Thinking of images as 2D arrays:

In [ ]:
```
import numpy as np

np.array(bw)
```

In [ ]:
```
np.array(gs)
```

STRONG NOTE: `dtype=uint8`

In [ ]:
```python
# Darken sections of the greyscale image via numpy array
ngs = np.array(gs)
ngs[:, :gs.width//2] //= 2  # [rows, columns]
ngs[:gs.height//4, :] //= 2  # [rows, columns]

display(Image.fromarray(ngs))
```

Blending Images

Two images can be **blended together** using a specified alpha (transparency) value using `Image.blend()`.

- Both images must share the same mode (e.g., RGBA) and dimensions.
- The `alpha` parameter controls the transparency of the blend.
  - 0 weights toward first, 1 toward second
- `Image.composite()` enable blending images using a mask.
  - A **mask** can selectively apply transformations.
  - For example, a circular mask can be created and composited with the original image.
    - The mask image should be in mode L (grayscale).
    - Creating smooth masks may require anti-aliasing techniques.
    - The `ImageFilter` module can *soften* the mask's edges for a smoother transition.

In [ ]:
```python
# Open two images of the same size
background = Image.open("./data/background.jpg")
foreground = Image.open("./data/foreground.png")

print(foreground.mode, background.mode)

# Ensure that both images have the same size and mode (conversion may be necessary)
foreground = foreground.resize(background.size)
foreground = foreground.convert("RGBA")
background = background.convert("RGBA")

# Blend images: an alpha value of 0.3 means the foreground is 40% opaque
blended = Image.blend(background, foreground, alpha=0.4)
blended.save("./data/blended_example.png")
display(blended)
```

In [ ]:
```python
# Create a mask image (grayscale) for advanced blending
mask = Image.new("L", background.size, 128)  # 50% transparency mask
composite = Image.composite(foreground, background, mask)
display(composite)
```

In [ ]:
```python
import math

# Create a new image for the mask with the same size as the original
mask = Image.new("L", image.size, 0)
mask_pixels = mask.load()

# Define a circle at the centre of the image
center_x, center_y = image.width // 2, image.height // 2
radius = min(center_x, center_y) - 10

for x in range(image.width):
    for y in range(image.height):
        # Compute the distance from the centre
        if math.sqrt((x - center_x) ** 2 + (y - center_y) ** 2) < radius:
            mask_pixels[x, y] = 255  # White pixel (mask active)
```

```
# Apply the mask to the image (assuming the image is in RGB mode)
masked_image = Image.composite(image, Image.new("RGB", image.size, "black"), mask)
display(masked_image)
```

In [ ]:
```
from PIL import ImageFilter

# Apply Gaussian blur to the mask to soften edges, value is radius in pixels
soft_mask = mask.filter(ImageFilter.GaussianBlur(50))
soft_masked_image = Image.composite(image, Image.new("RGB", image.size, "black"), soft_mask)
display(soft_masked_image)
```

Annotating and Drawing on Images

The `ImageDraw` module can be used to add **text** and draw **shapes** on an image.

```
from PIL import ImageDraw, ImageFont
```

- Specifying a custom font requires a valid **TTF** file.
- More sophisticated annotations, such as ellipses or semi-transparent overlays, can also be placed on a given image.

See more utilities: https://pillow.readthedocs.io/en/stable/reference/ImageDraw.html

- Free fonts can be downloaded from sites such as Google Fonts or DaFont.
  - For example: https://fonts.google.com/specimen/Big+Shoulders+Stencil

In [ ]:
```
from PIL import Image
from PIL import ImageDraw

# Create an RGBA copy of the image to draw on
img = Image.open("./data/example.jpg")

# Create an ImageDraw object to process draw elements
# NOTE: the image will be modified in place!
draw = ImageDraw.Draw(img, 'RGBA')  # specify mode

# Draw a red rectangle, provide [(x0, y0), (x1, y1)] or [x0, y0, x1, y1]
draw.rectangle(((250, 250), (550, 550)),
               fill=(30, 200, 200, 90), # Transparent fill
               outline='red',
               width=5)

display(img)
```

In [ ]:
```
from PIL import ImageDraw, ImageFont

# Create an RGBA copy of the image to draw on
annotated_image = Image.open("./data/example.jpg")

# Create an ImageDraw object to process draw elements
# NOTE: the image will be modified in place!
draw = ImageDraw.Draw(annotated_image, 'RGBA')  # specify mode

# Draw a red rectangle, provide [(x0, y0), (x1, y1)] or [x0, y0, x1, y1]
draw.rectangle(((250, 250), (550, 550)),
               fill=(30, 200, 200, 90), # Transparent fill
               outline='red',
               width=5)

# Annotate the image with text (default font)
```

```
draw.text((60, 1060), "Sample Annotation", fill="white", font_size=150)
display(annotated_image)
```

In [ ]:
```
# Configure the font details
font = ImageFont.truetype("./data/BigShouldersStencil-Regular.ttf", size=160)

# Note: bitmap fonts are loaded using ImageFont.load()

# Annotate the image with text (default font)
draw.text((60, 1460), "ANOTHER Annotation", fill="white", font=font)
display(annotated_image)
```

Introduction to OpenCV

OpenCV (Open Source Computer Vision Library, `cv2`) is a comprehensive library aimed at optimised computer vision and image processing.

While Pillow focuses on general image manipulation (reading, writing, transforming, and annotating images), OpenCV is designed for **more advanced tasks** such as computer vision, video capture, object detection, and real-time image processing.

- See: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

Comparison:

Use Case	Pillow	OpenCV
Basic Image I/O	Simple and intuitive API for reading and saving a wide range of image formats.	Supports more file formats including video capture and streaming.
Image Manipulation	Easy cropping, resizing, rotating, and basic transformations with high-level functions.	Extensive support for affine transformations, warping, and advanced geometric operations.
Image Filtering & Effects	Offers basic filters (e.g., blur, sharpen) and simple enhancements.	Provides advanced filtering techniques such as Canny edge detection, Sobel, and morphological operations.
Drawing & Annotation	Supports drawing text, lines, and shapes directly on images with a straightforward API.	Contains basic drawing functions mostly used for debugging and visualization.
Color Space Conversion	Handles common mode conversions (e.g., RGB, RGBA, L) efficiently.	Offers a broader range of conversions (e.g., HSV, LAB, YCrCb), beneficial for computer vision tasks.
Performance	Optimised for ease of use and suitable for small, off-line to medium-scale image processing tasks.	Highly optimized for real-time processing and handling large-scale images or video streams.
Computer Vision	Primarily focused on image processing rather than advanced vision tasks.	A comprehensive library designed for advanced computer vision and machine learning applications.
Ecosystem Integration	Easily integrates with Python libraries like NumPy and SciPy for general image processing.	Frequently used alongside deep learning frameworks and specialized computer vision libraries.
Video Processing	Limited to basic animated GIF handling; not intended for comprehensive video processing.	Extensive support for video capture, processing, and real-time streaming with built-in functions.

Reading and displaying

OpenCV reads images as **NumPy arrays**, which facilitates integration with numerical operations.

Note: OpenCV utilises `BGR` ordering instead of `RGB` (common in Pillow), which is an important consideration when processing colours.

In [ ]:
```
import cv2

# Read the image (reads in BGR format)
img = cv2.imread("./data/example_edge.jpg")
cv2.imshow("OpenCV Image", img)
cv2.waitKey(0)  # Wait until a key is pressed
cv2.destroyAllWindows()
```

Example: Edge Detection

Edge detection is a common computer vision task, and OpenCV's `Canny` algorithm computes edges efficiently.

In [ ]:
```
# onvert to greyscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply Canny edge detection
edges = cv2.Canny(gray, threshold1=50, threshold2=150)

# Display the result
cv2.imshow("Edges", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:
```
from matplotlib import pyplot as plt

plt.imshow(edges, cmap=plt.cm.binary_r, vmin=0, vmax=255)
plt.gca().set_axis_off()
plt.gca().set_aspect('equal')
```

Video Capture

OpenCV is capable of capturing video in real time, processing each frame (for example, converting to greyscale), and displaying it.

`VideoCapture`:

This real-time capability could be used for applications such as surveillance systems or interactive robotics.

In [ ]:
```
#  This generates a live video capture window
#  It can take a minute to become live.

cap = cv2.VideoCapture(0)

if not cap.isOpened():
    print("Cannot open camera")
    exit()

while True:
    # Capture frame-by-frame
    ret, frame = cap.read()
    if not ret:
        print("Can't receive frame. Exiting ...")
        break

    # Convert frame to greyscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Display the resulting frame
    cv2.imshow('Live Video - Greyscale', gray_frame)
    if cv2.waitKey(1) == ord('q'):
        break

# When everything is done, release the capture
cap.release()
cv2.destroyAllWindows()
```

In [ ]:
```
# Inspect the final frame
frame.shape
```

In [ ]:
```
# View the last capture
display(Image.fromarray(frame))
```

In [ ]:
```
# Adjust the colour order
display(Image.fromarray(frame[..., [2, 1, 0]]))
```

In [ ]:
```
# Mode conversion with cv2
image_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
plt.imshow(image_rgb)
```

In [ ]:
```
# Write image to file
cv2.imwrite('./data/frame.jpg', frame)
```

In [ ]:
```
# Apply Gaussian blur with a kernel (odd dimensions)
kernel = 39
blurred_img = cv2.GaussianBlur(frame, (kernel, kernel), 0)

cv2.imshow("Blurred Image", blurred_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Image processing summary

While **Pillow** provides an accessible and powerful toolkit for basic image manipulations and static image processing, **OpenCV** offers advanced computer vision and real-time video processing capabilities.

Both libraries have their respective roles within the Python ecosystem, and understanding when and how to use each (often in combination) allows a wide variety of image and video processing tasks to be addressed effectively.

More Pillow exploration: https://realpython.com/image-processing-with-the-python-pillow-library

---

MUCH Further Exploration

This is a list of many of the items that we would like to explore in this module but do not have the scope to reasonably include.

There is much, much more out there.

Linting (Black): Black Documentation

- A formatter that automatically reorders code to conform to a standard style.
- It assists in maintaining a consistent code style throughout a programme.

Linting (flake8): flake8 Documentation

- A tool that checks code for stylistic and logical errors according to community guidelines.
- It helps students identify and correct common issues to improve code readability.

Type Hinting: Python Type Hints (typing)

- A system that allows annotations of variable and function types to clarify code intentions.
- It supports better tooling and readability by making code easier to understand.

TypeGuard: Python TypeGuard Documentation

- A specialised feature that refines type checking by narrowing types based on runtime checks.
- It aids in developing more robust programmes by allowing more precise type validation.

Raising Warnings: Python Warnings

- A method to alert users of non-fatal issues within code execution.
- It encourages a mindful approach to handling potential problems without interrupting programme flow.

cProfile: cProfile Documentation

- A built-in profiler that measures the performance of different parts of a programme.
- It provides insights into code execution time, helping in identifying areas for improvement.

Security Modules (secrets): Python Secrets Documentation

- A module that offers functions for generating secure tokens and random numbers.
- It introduces students to basic practices for handling sensitive data in programmes.

Security Modules (hashlib): Python Hashlib Documentation

- A module that provides access to various hashing algorithms for data verification.
- It demonstrates how to implement hash functions for integrity and password management.

Security Modules (ssl): Python SSL Documentation

- A module that enables encrypted network communication through SSL/TLS protocols.
- It shows students how to secure data in transit over networks.

Widgets (ipywidgets): ipywidgets Documentation

- A library for adding interactive widgets to Jupyter Notebooks.
- It enhances the user experience in interactive computing environments.

Python to Cython: Cython Documentation

- A method for translating Python code to C for improved performance.
- It introduces an approach to optimise code execution by interfacing with lower-level languages.

Numba: Numba Documentation

- A just-in-time compiler that accelerates numerical computations in Python.
- It provides a way to improve performance in data-intensive tasks with minimal code changes.

`__new__` for Singletons and Extending Immutables: Python Data Model – **new**

- A technique for customizing object creation and managing immutable types.
- It offers insights into the inner workings of Python's object model and advanced design patterns.

Memory Management & Garbage Collection: Garbage Collection Documentation

- An overview of how Python manages memory and recovers unused resources.
- It is important for understanding performance and resource management in programmes.

Advanced Metaprogramming (Beyond Decorators): Metaclasses and Advanced Metaprogramming

- An exploration of techniques that modify programme behaviour during runtime beyond simple decorators.
- It enables students to write more flexible and adaptive code by understanding deeper programming constructs.

Richness of the Standard Library Python Standard Library Index

- A look into the wide range of modules available in the standard library that support common tasks.

- It highlights how built-in modules can simplify complex programming tasks and improve code clarity.

    - **itertools:** Python itertools Documentation

        - A module that provides functions for efficient looping and iteration.
        - It assists in simplifying complex iteration tasks in various programming scenarios.

    - **collections:** Python collections Documentation

        - A module that offers specialised container datatypes beyond the basic lists and dictionaries.
        - It supports more organised data management and storage in programmes.

    - **functools:** Python functools Documentation

        - A module that includes higher-order functions to work with callable objects and enhance function behaviour.
        - It helps in creating cleaner code through techniques like function composition and memoisation.

Packages: Python Packaging User Guide

- A collection of tools and practices for managing project-specific dependencies and environments.

- It assists in maintaining organised and isolated project configurations.

    - **Package Generation (including Namespace Packages):** Packaging Python Projects

        - A guide to organising and distributing Python code, including the use of namespace packages.
        - It familiarises learners with the process of sharing and reusing code in larger projects.
    - **venv:** Python venv Documentation
        - A built-in module for creating isolated virtual environments in a Python project.
        - It supports dependency isolation and helps prevent conflicts between projects.
    - **pipenv:** Pipenv Documentation
        - A tool that integrates virtual environment creation and dependency management in one interface.
        - It simplifies the process of maintaining project-specific dependencies.
    - **poetry:** Poetry Documentation
        - A package that provides a complete solution for dependency management and project packaging.
        - It offers a streamlined workflow for configuring and managing project environments.
    - **astral's uv:** uv Documentation
        - A tool for managing dependencies in some projects, new, **efficient** and growing.
        - It presents an alternative method for dependency management that certain projects may consider.

Interfacing with C Libraries Beyond Cython: ctypes Documentation

- A module that provides a way to call functions in dynamic libraries and interact with C code.
- It offers an alternative method for integrating C libraries to enhance programme performance.

In This Week's Practical

Practice with:

- The Walrus Operator for file reads
- Decorators for bespoke caching
- Dataclasses for book management
- Logging via email (simulated/real)
- Creating images in pillow from numpy arrays
- Green screen processing

Next Week

...is actually partially this week!

- Lecture is on a different **day**: *Friday 2nd May*.
- It is at a different **time of day**: *3pm*
- It will be held in a different **space**: *Palmer G10*.
- There will be a new **lecturer**: *Dr Martin Lester*

Developing Graphical User Interfaces (GUIs)