

20. Further Programming

20.1 Programming Paradigms

- A **programming paradigm** is a **fundamental style or approach** to programming.
 - It is a model that dictates how one **writes and organizes** codes.
- There are four paradigms to consider: **low-level**, **imperative (procedural)**, **object oriented**, and **declarative**.

Low-level Programming

- Example: machine code / assembly language.
- Languages which use the basic machine operations of the processor.
- Close to the architecture of the processor.
- Assembly language has a one-to-one mapping with machine code.
- Assembly language uses **mnemonics**.

Imperative (Procedural) Programming

- Imperative languages use **variables**
- ... Which are changed using (assignment) statements
- ... They rely on method of **repetition / iteration**.
- The statements can provide a sequence of commands to perform.
- ... In the order written / given
- ... Each line of code changes something in the program run.

Example

- All the Python codes that we've written before are **procedural**:

```
total = 0
for i in range(1, 6): # sum numbers from 1 to 5
    total += i
print("The sum is:", total)
```

- The program above is imperative because:
 - You explicitly **declare variables** (`total`).
 - You **manually control the flow** using a `for` loop.
 - You **update state** (`total += i`) with each iteration.
 - You **tell the computer what to do step-by-step**.

Declarative Programming

- Instructs a program on **what needs to be done** instead of how to do it
- ... Using **facts** and **rules**
 - **Facts**: an entity that is known.

- **Rules:** relationships between facts.
- ... Using **queries** to satisfy goals.
- Can be **logical** or **functional**.
 - **Logical:** states a program as a set of **logical relations**.
 - **Functional:** constructed by applying functions to arguments.

Example

- It is required by the syllabus to write the **prolog** language:

Declarative language reminder

- Always add a period "." after a declarative statement.

```
% define country "france," language "french," and their relation.
country(france).
language(french).
food(snail).
country_uses_language(france, french).
country_eats_food(france, snail)

% make queries.
country_uses_language(X, french). // returns "X = france"
country_eats_food(france, Y). // returns "Y = snail"
```

Object-Oriented Programming (OOP)

- Problem are modeled with **objects**.
- Objects are defined in **classes**.
- Objects contain both the **properties/data/attributes** and the **methods**.
- Properties can be read or written using **methods**.

OOP Terms

- **Inheritance.**
 - The capability of defining a new class of objects that has all the attributes and methods from a **parent class**.
- **Polymorphism.**
 - Allows the same method to take on different behaviors depending on which class is instantiated.
- **Containment (Aggregation).**
 - Where one class contains a reference to another class.
 - The contained object **can exist independently** of the container.
- **Encapsulation.**
 - The process of putting data and methods together as a single unit.
- **Getters.**
 - Methods used to return the value of a property.
- **Setters.**
 - Methods used to update the value of a property.

- **Instances.**
 - An occurrence of an object.

Example

- For the pseudocode syntax, please refer to the [pseudocode guide](#).

```
class Student: # object defined in class
    def __init__(self, age, gender): # constructor
        self.__age = age # private attribute
        self.__gender = gender # private attribute

    def grow_up(self): # method
        self.__age += 1

    def get_age(self): # getter
        return self.__age

    def set_age(self, age): # setter
        if isinstance(age, int):
            self.__age = age

class Advisor: # A separate class to be aggregated / contained
    def __init__(self, name, department):
        self.name = name
        self.department = department

    def get_info(self):
        return f"{self.name} from {self.department}"

class SCIEStudent(Student): # inheritance
    def __init__(self, age, gender, house, advisor):
        super().__init__(age, gender) # call parent constructor
        self.__house = house # new property
        self.advisor = advisor # aggregation: call other classes

    def grow_up(self): # override; polymorphism
        super().grow_up() # call parent method
        self.__house = "black prefect"

    def get_advisor_info(self):
        return self.advisor.get_info()

# Aggregated instance
mr_shawn = Advisor("Mr. Shawn Wu", "Computer Science")

# Create an instance
Hardy = SCIEStudent(18, "male", "water", mr_shawn)

# Example usage
print(Hardy.get_advisor_info()) # → Mr. Shawn Wu from Computer Science
```

20.2 File Processing and Exception Handling

File processing

- For syntax related to file processing, please refer to:
 - **Pseudocode:** pseudocode guide (pay attention to how to handle random files).
 - **Python:** [W3Schools](#).

Exception Handling

- An **exception** is an unexpected event that disrupts the execution of a program.
 - E.g., file not found, division by zero, etc.
- **Exception handling** is the process of responding to an exception within the program so that the program **does not halt unexpectedly**.

Reasons for Exception Handling

- To trap (some) **runtime** errors.
- To prevent a program halting unexpectedly.
- To produce meaningful error messages for these errors.
- Example: divide by zero / end of file / file not found.

Python Syntax

- The `try...except...` control flow statement is used in Python to handle exceptions. You shall understand the example below for A-Level exams:

```
def divide(x, y):
    try:
        # try the division
        result = x / y
        print("Result:", result)
    except:
        # catch any kind of exception (not recommended in large programs)
        print("Something went wrong!")

# example 1: valid input
divide(10, 2)    # Output: Result: 5.0

# example 2: division by zero
divide(10, 0)    # Output: Something went wrong!

# example 3: invalid input type
divide(10, "a")  # Output: Something went wrong!
```

- However, it is recommended to specify the specific errors to be caught in `except` statements, and to write a more robust exception handling pipeline.
- An example is given below:

```
def divide(x, y):
    try:
        # try to execute this block – risky operation
```

```
    result = x / y
except ZeroDivisionError:
    # this block runs if a division by zero is attempted
    print("Error: You can't divide by zero.")
except TypeError:
    # this block runs if a non-numeric type is used in division
    print("Error: Please enter numbers only.")
else:
    # this block runs only if no exceptions were raised
    print("Result:", result)
finally:
    # this block always runs, no matter what happened above
    print("This block always runs.")

# example 1: valid division
divide(10, 2)    # output: Result: 5.0

# example 2: division by zero – triggers ZeroDivisionError
divide(10, 0)    # output: Error message for division by zero

# example 3: invalid input type – triggers TypeError
divide(10, "a")  # output: Error message for non-numeric input
```