Homework4

黄家睿(Jerry Huang)
202283890036

In the previous lesson, we learned about features such as class encapsulation, inheritance, polymorphism, regular expressions, exception handling, and code introspection.
In this assignment, we will apply these new features to enhance the evolutionary algorithm class we previously implemented.

encapsulation

Encapsulate the core logic, parameter initialization, and data processing operations of the algorithm within a class, ensuring that external code can only access the core functionalities through the interfaces provided by the class.

In this assignment, we will encapsulate the parameters and methods of the BWO algorithm within a class, making the code more structured.

```
class BWO_CLASS:
    def __init__(self, population_size=300, max_iterations=500,
     objective_function="F9"):

......


        print(str(self.objective_function) + " " +
              str(self.max_iterations) + " " +
              str(self.population_size) + " " +
              str(ub) + " " +
              str(lb) + " " +
              str(dim) + " " +
              str(fobj))
```

inheritance and polymorphism

Design the parent class to implement a general algorithm framework, while the subclasses override certain methods based on different optimization requirements or objective functions, achieving polymorphism.

The subclass `BWO_solution` inherits from the base class `BWO_CLASS` and overrides the `BWO_function` to implement different behaviors.

```
class BWO_CLASS:

......

def BWO_function(self):
```

```
        func_details1 = benchmarks.getFunctionSet(self.objective_function)
        lb = func_details1[1]
        ub = func_details1[2]
        dim = func_details1[3]
        fobj = getattr(benchmarks, self.objective_function)  # get the solution of
the function
        print(str(self.objective_function) + " " +
              str(self.max_iterations) + " " +
              str(self.population_size) + " " +
              str(ub) + " " +
              str(lb) + " " +
              str(dim) + " " +
              str(fobj))


#The subclass `BWO_solution` overrides the `BWO_function` method.

class BWO_solution(BWO_CLASS):
    def BWO_function(self):
        func_details1 = benchmarks.getFunctionSet(self.objective_function)
        lb = func_details1[1]
        ub = func_details1[2]
        dim = func_details1[3]
        fobj = getattr(benchmarks, self.objective_function)  # get the solution of
the function
        x = BWO.BWO(fobj, lb, ub, dim, self.population_size, self.max_iterations)
```

regular expressions

During the initialization process, regular expressions are used to validate the legality of input parameters, such as checking whether the objective function name adheres to naming conventions.

In this assignment, the regular expression is used to validate that the `objective_function` parameter contains only valid characters.

```
# Use regular expressions to validate whether the objective function name conforms
to the specified naming conventions.
        if not re.match(r"^[A-Za-z0-9_]+$", objective_function):
            raise ValueError("The objective function name contains illegal
characters!")
        self.objective_function = objective_function
```

exception handling

Add exception handling mechanisms to the key logic sections to catch and handle potential runtime errors, preventing the program from crashing while providing meaningful error messages.

The addition of exception handling helps us capture exceptions in function calls and attribute accesses, offering clear error messages.

```
            if fobj is None:
                raise AttributeError(f"target
function{self.objective_function}does not find in benchmarks Module! ")

            print(f"target function: {self.objective_function}, Maximum
iterations: {self.max_iterations}, Population size: {self.population_size}")
            print(f"search range: [{lb}, {ub}], Dimension: {dim}")
        except Exception as e:
            print(f"An error occurred while executing the `BWO_function`.: {e}")
```

code introspection

Implement code introspection functionality to dynamically check the attributes and methods of a class and its objects, allowing for debugging and validation of whether the class implementation meets expectations.

This functionality is achieved through the `inspect_class` function, which dynamically inspects the attributes and methods of the class.

```
def inspect_class(cls):
    print(f"check class: {cls.__name__}")
    print(f"function list: {[method for method in dir(cls) if
callable(getattr(cls, method)) and not method.startswith('__')]}\n")
```