Evaluation Result by example problems in github:

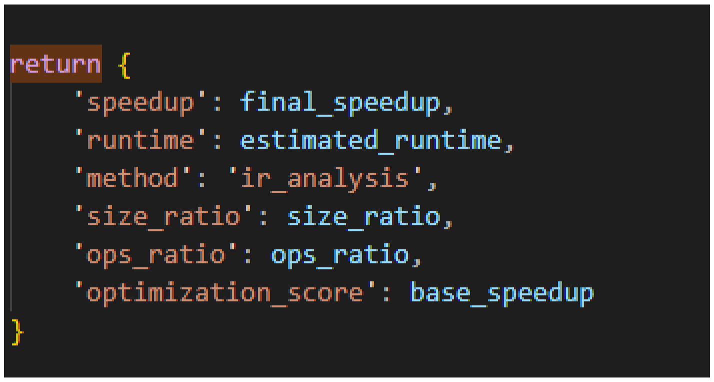
Conclusion after Collection of data:

After looking over the given examples, my ideas of how to actually utilize the Open Evolve framework, especially in creating evaluator.py scripts, has really opened up. It seems that the cleanest and intended practice is to use the given class by Open Evolve of EvaluationResult. However like all programs many developers just seemed to have duck taped it together by skipping past intended conventions in many examples.

I can only conclude that the best practice is our practice, as in the developer intending to use the system.

Attention Optimization:

The attention optimization problem also returns a raw dictionary but calls the score a different name

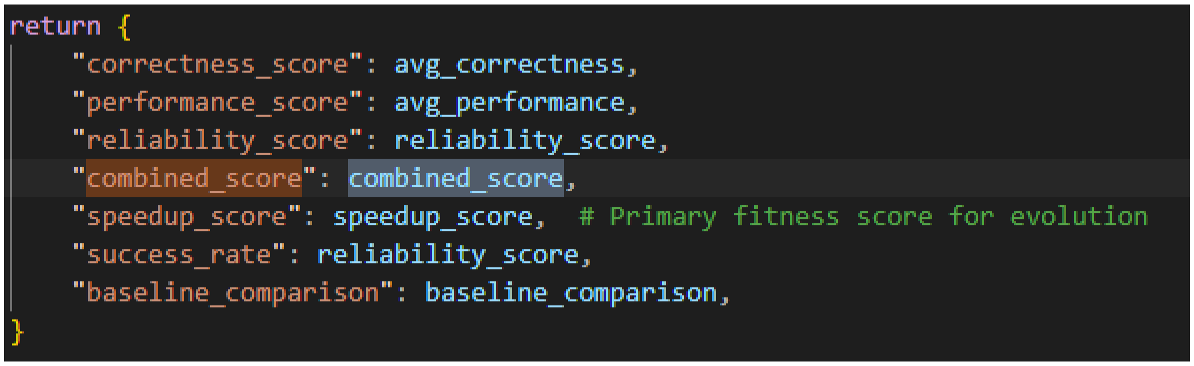


```
return {
    'speedup': final_speedup,
    'runtime': estimated_runtime,
    'method': 'ir_analysis',
    'size_ratio': size_ratio,
    'ops_ratio': ops_ratio,
    'optimization_score': base_speedup
}
```

Algotune

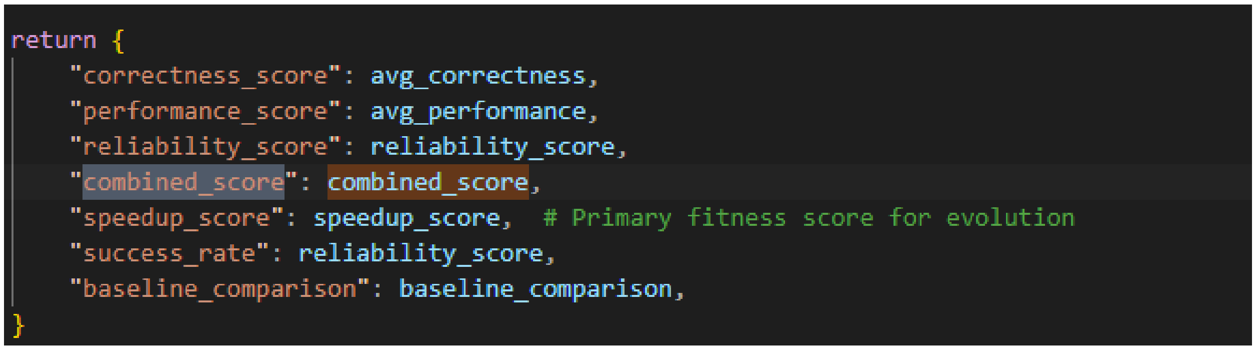
All the algotune examples are consistent in their return types, containing correctness, performance, combined,etc. scores.

Affine Transformation 2d:



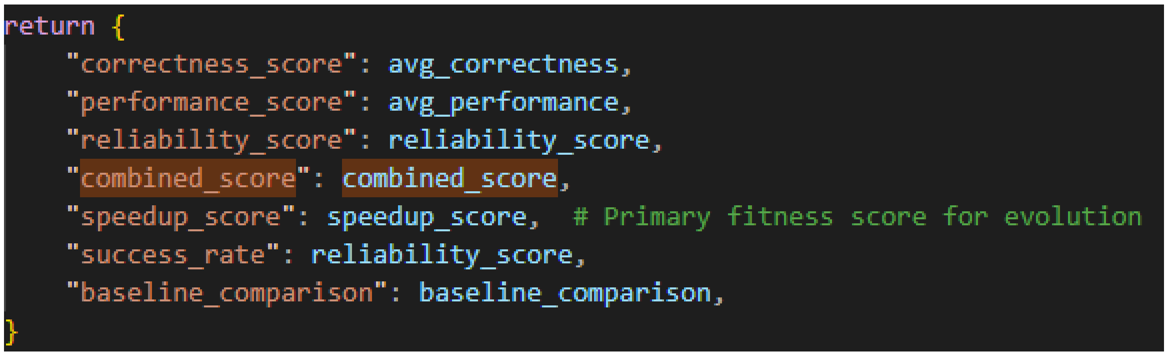
```
return {
    "correctness_score": avg_correctness,
    "performance_score": avg_performance,
    "reliability_score": reliability_score,
    "combined_score": combined_score,
    "speedup_score": speedup_score,  # Primary fitness score for evolution
    "success_rate": reliability_score,
    "baseline_comparison": baseline_comparison,
}
```

Convolve 2d Full Fill:



```
return {
    "correctness_score": avg_correctness,
    "performance_score": avg_performance,
    "reliability_score": reliability_score,
    "combined_score": combined_score,
    "speedup_score": speedup_score,  # Primary fitness score for evolution
    "success_rate": reliability_score,
    "baseline_comparison": baseline_comparison,
}
```

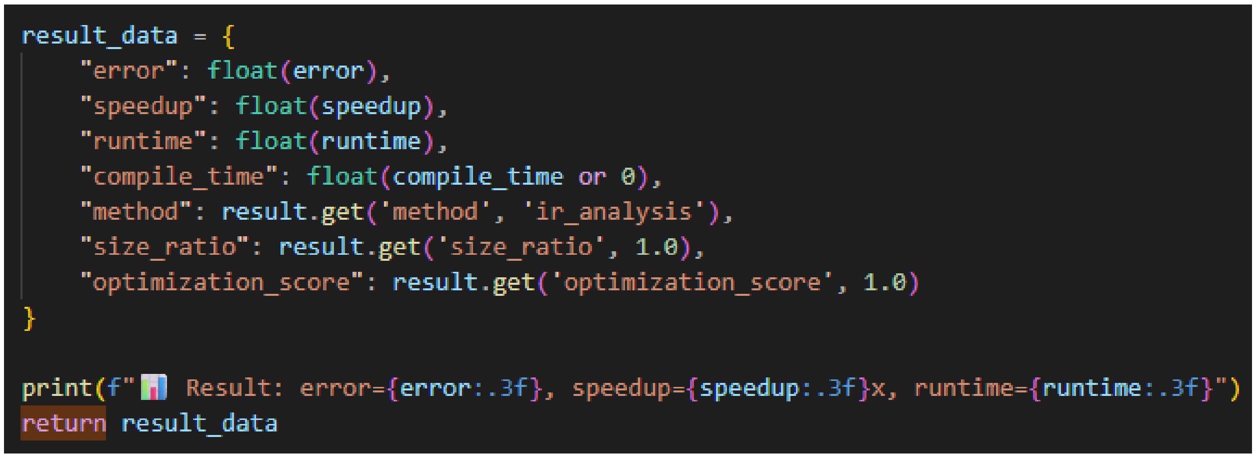
Eigen Vectors:



```
return {
    "correctness_score": avg_correctness,
    "performance_score": avg_performance,
    "reliability_score": reliability_score,
    "combined_score": combined_score,
    "speedup_score": speedup_score,  # Primary fitness score for evolution
    "success_rate": reliability_score,
    "baseline_comparison": baseline_comparison,
}
```

Attention Optimization:

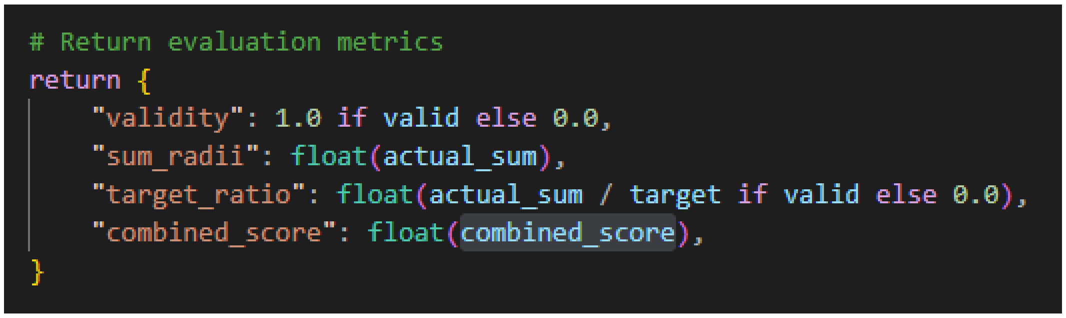
Another one that does not use combined score nor the actual data class type



```python
result_data = {
    "error": float(error),
    "speedup": float(speedup),
    "runtime": float(runtime),
    "compile_time": float(compile_time or 0),
    "method": result.get('method', 'ir_analysis'),
    "size_ratio": result.get('size_ratio', 1.0),
    "optimization_score": result.get('optimization_score', 1.0)
}

print(f"📊 Result: error={error:.3f}, speedup={speedup:.3f}x, runtime={runtime:.3f}")
return result_data
```

Circle Packing:

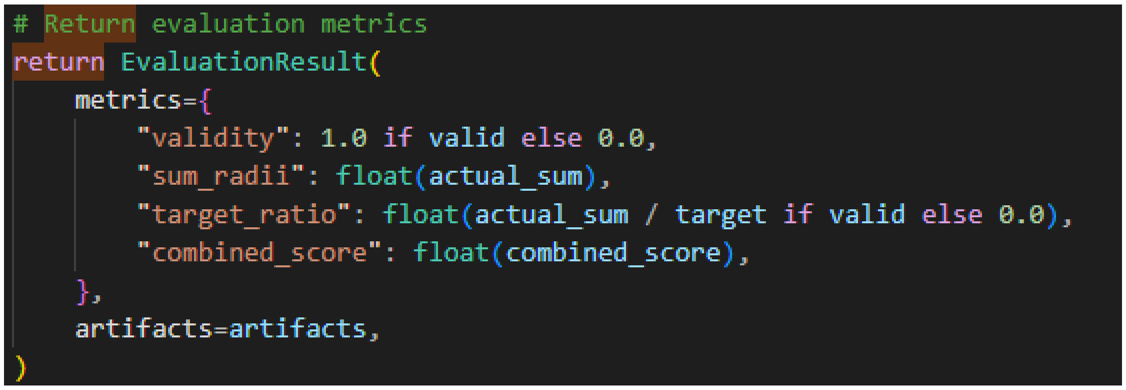
The circle packing problem just returns a raw dictionary for its metrics



```python
# Return evaluation metrics
return {
    "validity": 1.0 if valid else 0.0,
    "sum_radii": float(actual_sum),
    "target_ratio": float(actual_sum / target if valid else 0.0),
    "combined_score": float(combined_score),
}
```

Circle Packing with Artifacts:

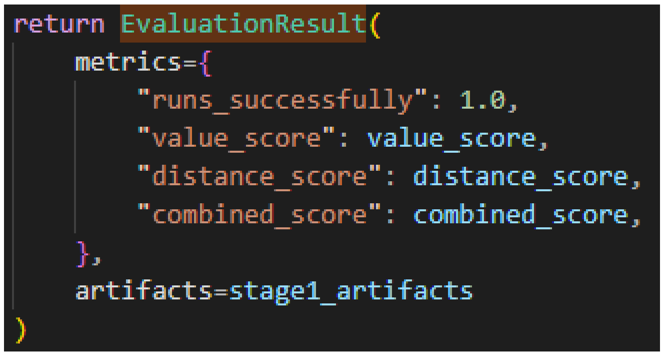
The circle packing with artifacts changes from the original to actually use the EvaluationResult class



```python
# Return evaluation metrics
return EvaluationResult(
    metrics={
        "validity": 1.0 if valid else 0.0,
        "sum_radii": float(actual_sum),
        "target_ratio": float(actual_sum / target if valid else 0.0),
        "combined_score": float(combined_score),
    },
    artifacts=artifacts,
)
```

Function Minimization:

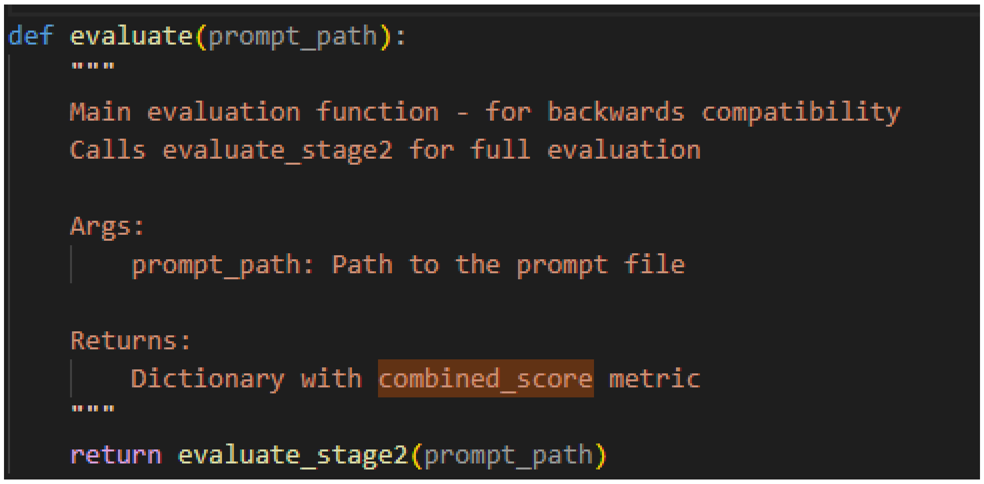
The function minimization uses the Evaluation result class provided by open evolve itself



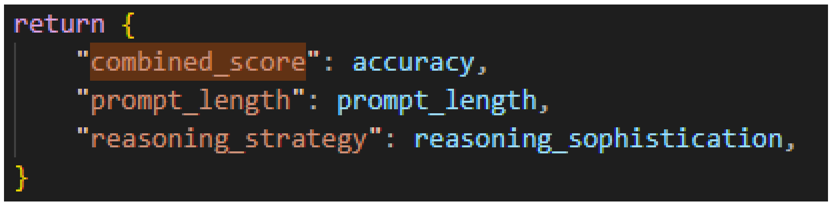
```
return EvaluationResult(
    metrics={
        "runs_successfully": 1.0,
        "value_score": value_score,
        "distance_score": distance_score,
        "combined_score": combined_score,
    },
    artifacts=stage1_artifacts
)
```

LLM Prompt Optimization:

Because this example specifically looks to evaluate itself with LLMs, there are multiple stages of evaluation (2) but they do use the combined score value.



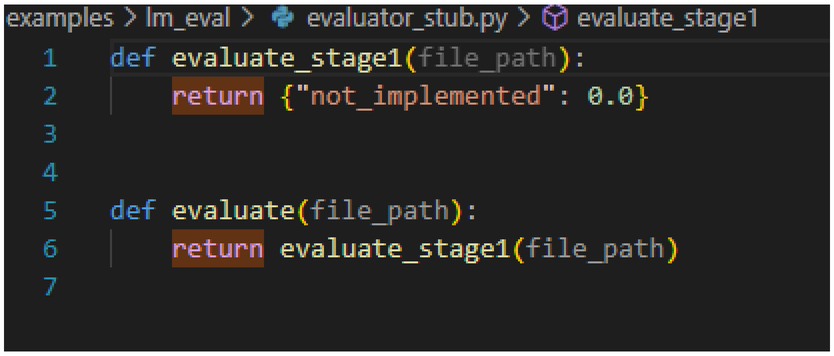
```
def evaluate(prompt_path):
    """
    Main evaluation function - for backwards compatibility
    Calls evaluate_stage2 for full evaluation

    Args:
        prompt_path: Path to the prompt file

    Returns:
        Dictionary with combined_score metric
    """
    return evaluate_stage2(prompt_path)
```



```
return {
    "combined_score": accuracy,
    "prompt_length": prompt_length,
    "reasoning_strategy": reasoning_sophistication,
}
```

LM Eval:

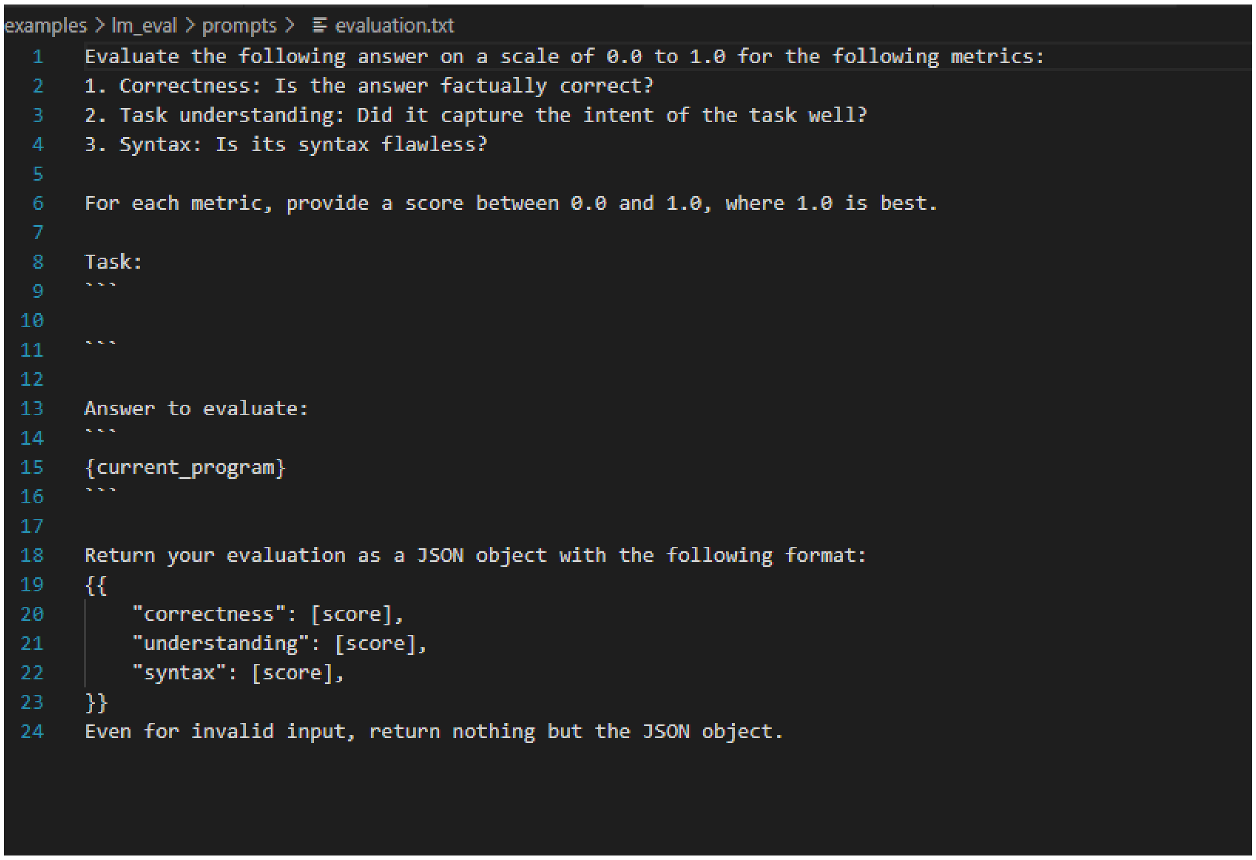
This one seems to be incomplete at least according to the evaluator stub.py file:



examples > lm_eval > evaluator_stub.py > evaluate_stage1

```python
def evaluate_stage1(file_path):
    return {"not_implemented": 0.0}


def evaluate(file_path):
    return evaluate_stage1(file_path)

```

Although they do seem to make up for it given the evaluation text:



examples > lm_eval > prompts > evaluation.txt

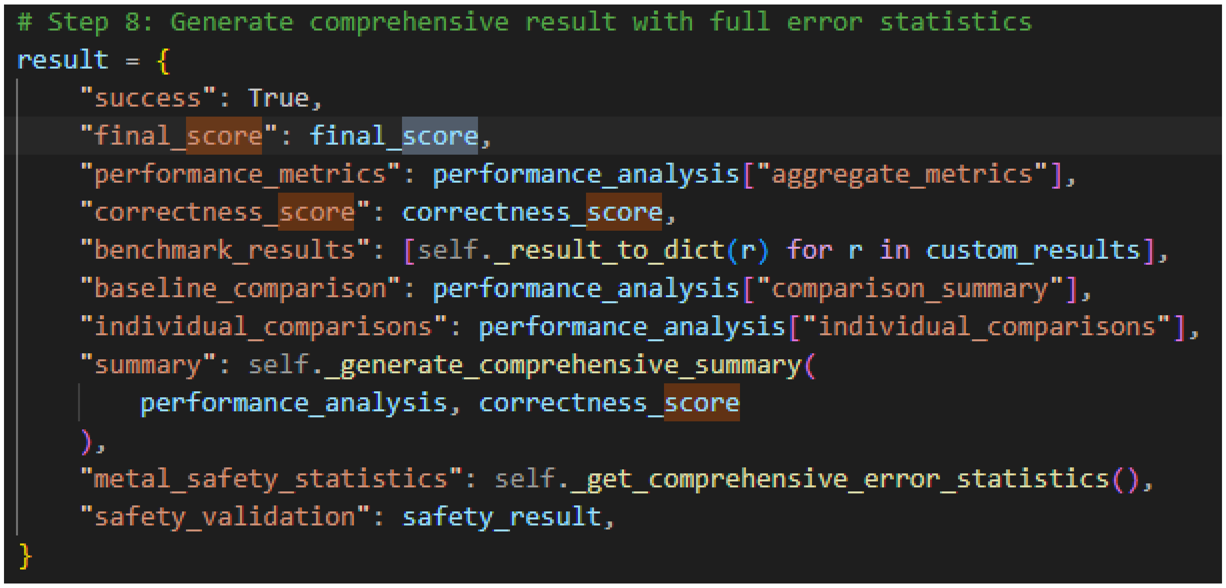
```
Evaluate the following answer on a scale of 0.0 to 1.0 for the following metrics:
1. Correctness: Is the answer factually correct?
2. Task understanding: Did it capture the intent of the task well?
3. Syntax: Is its syntax flawless?

For each metric, provide a score between 0.0 and 1.0, where 1.0 is best.

Task:
```

```

Answer to evaluate:
```
{current_program}
```

Return your evaluation as a JSON object with the following format:
{{
    "correctness": [score],
    "understanding": [score],
    "syntax": [score],
}}
Even for invalid input, return nothing but the JSON object.
```

Which also does not follow the general convention of combined_score or the Evaluation Result.

MLX Metal Kernel Opt:

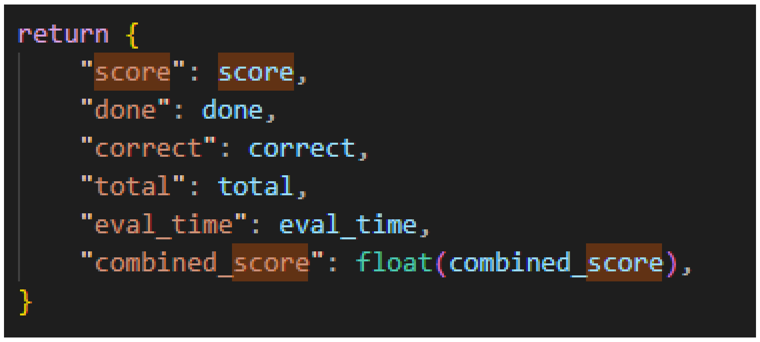
This one uses a “final_score” it seems instead of “combined_score”



```
# Step 8: Generate comprehensive result with full error statistics
result = {
    "success": True,
    "final_score": final_score,
    "performance_metrics": performance_analysis["aggregate_metrics"],
    "correctness_score": correctness_score,
    "benchmark_results": [self._result_to_dict(r) for r in custom_results],
    "baseline_comparison": performance_analysis["comparison_summary"],
    "individual_comparisons": performance_analysis["individual_comparisons"],
    "summary": self._generate_comprehensive_summary(
        performance_analysis, correctness_score
    ),
    "metal_safety_statistics": self._get_comprehensive_error_statistics(),
    "safety_validation": safety_result,
}
```

Online Judge Programming:

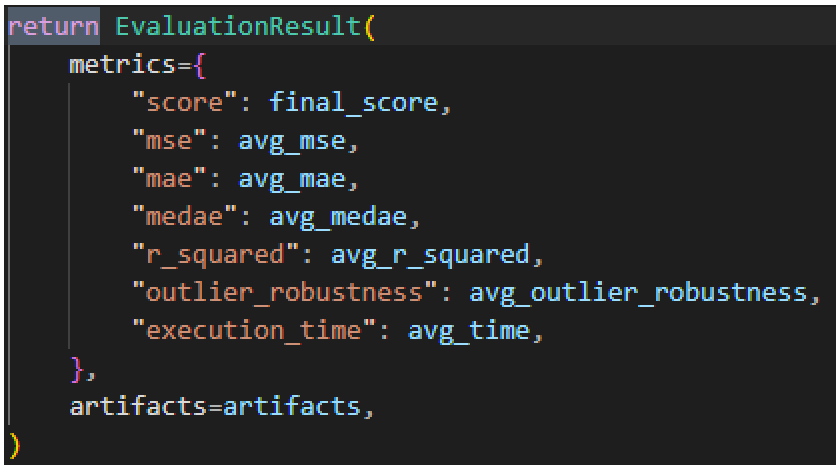
This one seems to be the most general, boasting its ability to compete in online coding judge challenges, but it also doesn't use the EvaluationResult data type.



```
return {
    "score": score,
    "done": done,
    "correct": correct,
    "total": total,
    "eval_time": eval_time,
    "combined_score": float(combined_score),
}
```

R Robust Regression:

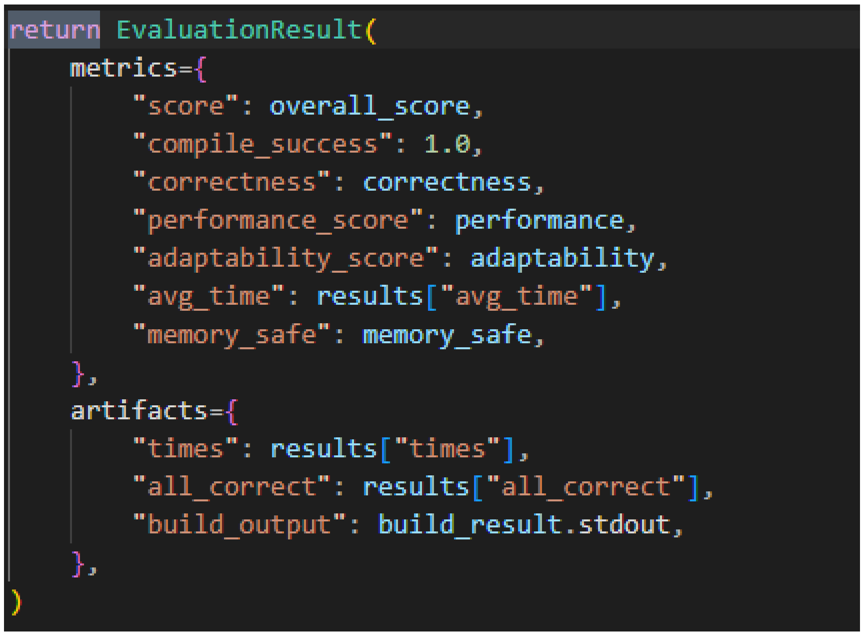
This one does use the data type, but interestingly enough doesn't use the combined score key for its metrics of “score”



```python
return EvaluationResult(
    metrics={
        "score": final_score,
        "mse": avg_mse,
        "mae": avg_mae,
        "medae": avg_medae,
        "r_squared": avg_r_squared,
        "outlier_robustness": avg_outlier_robustness,
        "execution_time": avg_time,
    },
    artifacts=artifacts,
)
```

Rust Adaptive Sort:

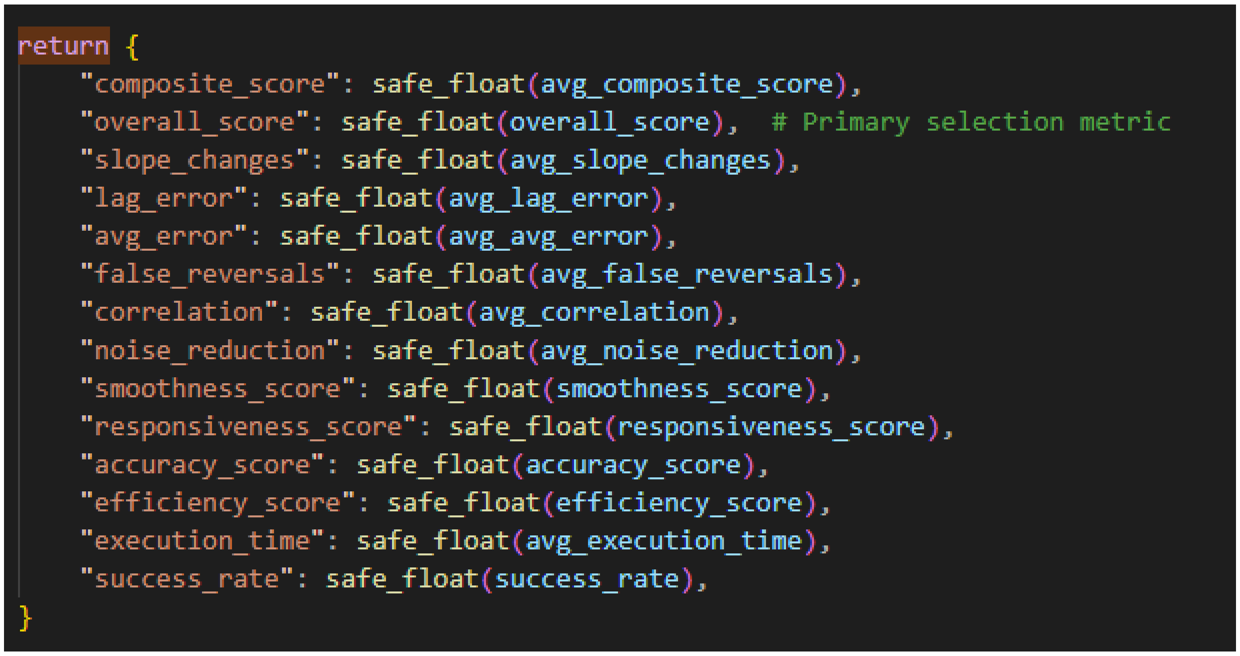
Like the Robust Regression, uses the class, not the key word.



```python
return EvaluationResult(
    metrics={
        "score": overall_score,
        "compile_success": 1.0,
        "correctness": correctness,
        "performance_score": performance,
        "adaptability_score": adaptability,
        "avg_time": results["avg_time"],
        "memory_safe": memory_safe,
    },
    artifacts={
        "times": results["times"],
        "all_correct": results["all_correct"],
        "build_output": build_result.stdout,
    },
)
```

Signal Processing:

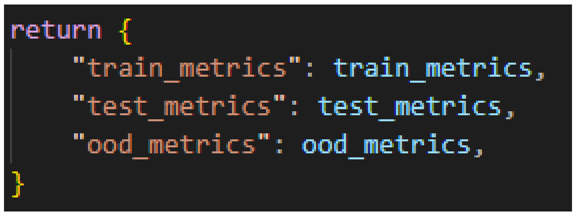
We are back to just the dictionary



```
return {
    "composite_score": safe_float(avg_composite_score),
    "overall_score": safe_float(overall_score),  # Primary selection metric
    "slope_changes": safe_float(avg_slope_changes),
    "lag_error": safe_float(avg_lag_error),
    "avg_error": safe_float(avg_avg_error),
    "false_reversals": safe_float(avg_false_reversals),
    "correlation": safe_float(avg_correlation),
    "noise_reduction": safe_float(avg_noise_reduction),
    "smoothness_score": safe_float(smoothness_score),
    "responsiveness_score": safe_float(responsiveness_score),
    "accuracy_score": safe_float(accuracy_score),
    "efficiency_score": safe_float(efficiency_score),
    "execution_time": safe_float(avg_execution_time),
    "success_rate": safe_float(success_rate),
}
```

Symbolic Regression:

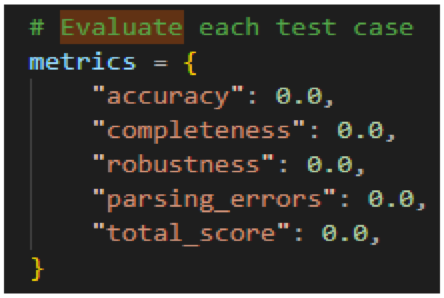
This one uses neither of the given conventions.



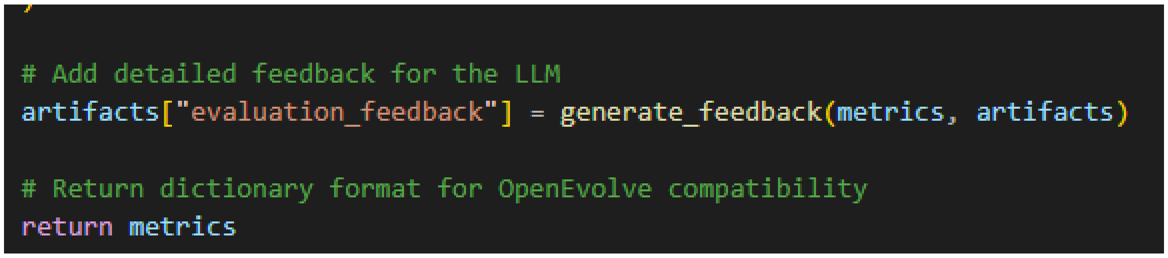
```
return {
    "train_metrics": train_metrics,
    "test_metrics": test_metrics,
    "ood_metrics": ood_metrics,
}
```

Web Scraper OptiLLM:

This one fills out and returns the metrics themselves while also keeping track of the artifacts, starting with an empty return value.



```python
# Evaluate each test case
metrics = {
    "accuracy": 0.0,
    "completeness": 0.0,
    "robustness": 0.0,
    "parsing_errors": 0.0,
    "total_score": 0.0,
}
```



```python

# Add detailed feedback for the LLM
artifacts["evaluation_feedback"] = generate_feedback(metrics, artifacts)

# Return dictionary format for OpenEvolve compatibility
return metrics
```