AutoGrader Complete Documentation

A comprehensive Python autograder for checking student code submissions. Supports variable validation, function testing, plot verification, control structure checking, list/array comparison, solution file comparison, and more.

Table of Contents

Installation

Required packages:

```bash
pip install numpy matplotlib pandas openpyxl reportlab
```

Quick Start

```python
```

```python
from autograder import AutoGrader

# Initialize grader with student's file
grader = AutoGrader("student_submission.py")

# Execute the script
grader.execute_script()

# Check variables
grader.check_variable_value('result', 42)

# Check if they used a for loop
grader.check_for_loop_used()

# Check list equality
grader.check_list_equals('my_list', [1, 2, 3, 4, 5])

# Compare with solution
grader.compare_with_solution('solutions/hw1.py', ['result', 'total'])

# Print summary with score
grader.print_summary()
```

Initialization

AutoGrader(filepath=None, timeout=10)

Initialize the autograder.

Parameters:

- `filepath` (str, optional): Path to the student's Python file
- `timeout` (int): Maximum execution time in seconds (default: 10)

Example:

```python
# Initialize with a file
grader = AutoGrader("student_code.py")

# Initialize with custom timeout
grader = AutoGrader("student_code.py", timeout=30)
```

Script Execution

`execute_script(variables_to_capture=None)`

Execute the student's entire script and capture variables.

Parameters:

- `variables_to_capture` (list, optional): List of variable names to capture. If `None`, captures all variables.

Returns: `bool` - `True` if execution succeeded, `False` otherwise

Example:

```python
```

```
# Execute and capture all variables
grader.execute_script()

# Execute and capture specific variables only
grader.execute_script(['x', 'y', 'total'])
```

Variable Checking

`check_variable_value(var_name, expected_value, tolerance=1e-6)`

Check if a variable has the expected value after execution.

Parameters:

- `var_name` (str): Name of the variable
- `expected_value` (any): Expected value
- `tolerance` (float): Tolerance for floating-point comparisons (default: 1e-6)

Returns: `bool` - `True` if check passes

Example:

```
python
```

```
# Check integer value
grader.check_variable_value('count', 10)

# Check float value with tolerance
grader.check_variable_value('average', 15.5, tolerance=0.1)

# Check string value
grader.check_variable_value('message', "Hello, World!")

# Check list value
grader.check_variable_value('numbers', [1, 2, 3, 4, 5])
```

check_variable_type(var_name, expected_type)

Check if a variable has the expected type.

Parameters:

- var_name (str): Name of the variable
- expected_type (type): Expected type (e.g., int, str, list)

Returns: bool - True if check passes

Example:

```
python

grader.check_variable_type('count', int)
grader.check_variable_type('message', str)
grader.check_variable_type('data', list)
```

List and Array Checking

check_list_equals(var_name, expected_list, order_matters=True, tolerance=1e-6)

Check if a variable contains a list equal to the expected list.

Parameters:

- `var_name` (str): Name of the variable
- `expected_list` (list): Expected list
- `order_matters` (bool): If `True`, order must match. If `False`, only elements need to match (default: `True`)
- `tolerance` (float): Tolerance for numeric comparisons (default: 1e-6)

Returns: `bool` - `True` if check passes

Works with: Lists, tuples, and NumPy arrays

Example:

```python
```

```
# Exact match with order
grader.check_list_equals('numbers', [1, 2, 3, 4, 5])

# Elements match, order doesn't matter
grader.check_list_equals('unsorted', [5, 3, 1, 4, 2], order_matters=False)

# Floating point list with tolerance
grader.check_list_equals('results', [1.5, 2.5, 3.5], tolerance=0.01)

# Works with NumPy arrays
grader.check_list_equals('np_array', [1.0, 2.0, 3.0])
```

`check_array_equals(var_name, expected_array, tolerance=1e-6)`

Check if a variable contains a NumPy array equal to the expected array.

Parameters:

- `var_name` (str): Name of the variable
- `expected_array` (array-like): Expected array (can be list or numpy array)
- `tolerance` (float): Tolerance for numeric comparisons (default: 1e-6)

Returns: `bool` - `True` if check passes

Features:

- Automatically converts lists to arrays
- Checks array shape (dimensions must match)
- Element-wise comparison with tolerance
- Works with multi-dimensional arrays

Example:

```python
# 1D array
grader.check_array_equals('vector', [1, 2, 3, 4, 5])

# 2D array
grader.check_array_equals('matrix', [[1, 2], [3, 4], [5, 6]])

# With tolerance for floating point
grader.check_array_equals('measurements', [1.0, 2.0, 3.0], tolerance=0.01)

# Works with both lists and numpy arrays
import numpy as np
grader.check_array_equals('data', np.array([1.5, 2.5, 3.5]))
```

Solution File Comparison

`compare_with_solution(solution_file, variables_to_compare, tolerance=1e-6)`

Execute a solution file and compare specified variables with student's values.

Parameters:

- `solution_file` (str): Path to the solution Python file
- `variables_to_compare` (list): List of variable names to compare
- `tolerance` (float): Tolerance for numeric comparisons (default: 1e-6)

Returns: `bool` - `True` if all variables match

How It Works:

1. Executes the student's code (already done)
2. Executes the solution file
3. Compares each specified variable between student and solution
4. Reports matches/mismatches for each variable

Features:

- Automatically handles different data types (numbers, lists, arrays, dicts)
- Uses appropriate comparison logic for each type
- Works with NumPy arrays
- Provides detailed feedback on which variables match/differ

Example:

python

```python
# Compare key variables with solution
grader.compare_with_solution(
    solution_file='solutions/assignment1.py',
    variables_to_compare=['result', 'total', 'average'],
    tolerance=0.01
)

# Compare multiple variables
grader.compare_with_solution(
    'solutions/hw2.py',
    ['x', 'y', 'z', 'final_answer']
)
```

Solution File Example:

```python
# solutions/assignment1.py
data = [10, 20, 30, 40, 50]
total = sum(data)
average = total / len(data)
result = average * 2
```

Folder Structure:

```
project/
├── autograder.py
├── student_submission.py
└── solutions/
    ├── assignment1.py
    ├── assignment2.py
    └── assignment3.py
```

Function Checking

`check_function_exists(func_name)`

Check if a function is defined in the code (static analysis).

Parameters:

- `func_name` (str): Name of the function

Returns: `bool` - `True` if function exists

Example:

```python
grader.check_function_exists('calculate_average')
grader.check_function_exists('process_data')
```

`check_function_called(func_name)`

Check if a function is called in the code. Supports module functions.

Parameters:

- `func_name` (str): Name of the function (can include module, e.g., 'np.mean', 'plt.plot')

Returns: `bool` - `True` if function is called

Example:

```python
# Check simple function call
grader.check_function_called('print')

# Check numpy function
grader.check_function_called('np.mean')
grader.check_function_called('numpy.std')

# Check matplotlib function
grader.check_function_called('plt.plot')
grader.check_function_called('plt.xlabel')

# Check nested module function
grader.check_function_called('numpy.random.randint')
```

Function Testing

test_function(func_name, test_cases)

Test a function with multiple test cases.

Parameters:

- `func_name` (str): Name of the function to test
- `test_cases` (list): List of test case dictionaries

Test Case Dictionary Keys:

- `args` (list): Positional arguments
- `kwargs` (dict, optional): Keyword arguments
- `expected` (any): Expected return value
- `tolerance` (float, optional): Tolerance for numeric comparisons

Returns: `bool` - `True` if all test cases pass

Example:

```python
```

```python
# Test a simple function
grader.test_function('add_numbers', [
    {'args': [5, 3], 'expected': 8},
    {'args': [10, 20], 'expected': 30},
    {'args': [0, 0], 'expected': 0}
])

# Test with keyword arguments
grader.test_function('power', [
    {'args': [2], 'kwargs': {'exp': 3}, 'expected': 8},
    {'args': [5], 'kwargs': {'exp': 2}, 'expected': 25}
])

# Test with floating-point tolerance
grader.test_function('calculate_average', [
    {'args': [[1, 2, 3]], 'expected': 2.0, 'tolerance': 0.01},
    {'args': [[10, 20, 30]], 'expected': 20.0, 'tolerance': 0.01}
])
```

Plot Validation

check_plot_created()

Check if any plot was created.

Returns: bool - True if plot exists

Example:

```python
```

```
grader.check_plot_created()
```

check_plot_properties(title=None, xlabel=None, ylabel=None, has_legend=None, has_grid=None, fig_num=1)

Check various properties of a matplotlib plot.

Parameters:

- `title` (str, optional): Expected plot title
- `xlabel` (str, optional): Expected x-axis label
- `ylabel` (str, optional): Expected y-axis label
- `has_legend` (bool, optional): Whether plot should have a legend
- `has_grid` (bool, optional): Whether plot should have a grid
- `fig_num` (int): Figure number to check (default: 1)

Returns: `bool` - `True` if all specified checks pass

Example:

```python
```

```python
# Check all properties
grader.check_plot_properties(
    title='Temperature Over Time',
    xlabel='Time (hours)',
    ylabel='Temperature (°C)',
    has_legend=True,
    has_grid=True
)

# Check only specific properties
grader.check_plot_properties(title='My Plot', has_legend=True)
```

`check_plot_data(expected_x=None, expected_y=None, line_index=0, tolerance=1e-6, fig_num=1)`

Check the data plotted in a line plot.

Parameters:

- `expected_x` (list, optional): Expected x-axis data
- `expected_y` (list, optional): Expected y-axis data
- `line_index` (int): Which line to check if multiple lines (default: 0)
- `tolerance` (float): Tolerance for numerical comparison (default: 1e-6)
- `fig_num` (int): Figure number to check (default: 1)

Returns: `bool` - `True` if data matches

Example:

```
python
```

```
# Check both x and y data
grader.check_plot_data(
    expected_x=[1, 2, 3, 4, 5],
    expected_y=[2, 4, 6, 8, 10]
)

# Check only y data
grader.check_plot_data(expected_y=[10, 20, 30, 40])

# Check second line in plot
grader.check_plot_data(expected_y=[5, 10, 15], line_index=1)
```

`check_plot_data_length(min_length=None, max_length=None, exact_length=None, line_index=0, fig_num=1)`

Check the length of the plotted data.

Parameters:

- `min_length` (int, optional): Minimum required data points
- `max_length` (int, optional): Maximum allowed data points
- `exact_length` (int, optional): Exact number of required data points
- `line_index` (int): Which line to check (default: 0)
- `fig_num` (int): Figure number to check (default: 1)

Returns: `bool` - `True` if length requirements are met

Example:

```
python
```

```
# Check minimum length
grader.check_plot_data_length(min_length=100)

# Check exact length
grader.check_plot_data_length(exact_length=50)

# Check range
grader.check_plot_data_length(min_length=10, max_length=100)
```

check_plot_function(function, line_index=0, tolerance=1e-6, fig_num=1)

Check if Y data is the correct function of X data.

Parameters:

- `function` (callable): A function that takes X data and returns expected Y data
- `line_index` (int): Which line to check (default: 0)
- `tolerance` (float): Tolerance for numerical comparison (default: 1e-6)
- `fig_num` (int): Figure number to check (default: 1)

Returns: `bool` - `True` if Y = function(X) within tolerance

Example:

```
python
```

```
import numpy as np

# Check if Y = cos(X * 2)
grader.check_plot_function(lambda x: np.cos(x * 2))

# Check if Y = X^2 + 3
grader.check_plot_function(lambda x: x**2 + 3)

# Check if Y = 2*X with custom tolerance
grader.check_plot_function(lambda x: 2*x, tolerance=0.01)
```

check_plot_color(expected_color, line_index=0, fig_num=1)

Check the color of a plotted line.

Parameters:

- `expected_color` (str): Expected color (e.g., 'red', 'r', '#FF0000', 'blue')
- `line_index` (int): Which line to check (default: 0)
- `fig_num` (int): Figure number to check (default: 1)

Returns: `bool` - `True` if color matches

Example:

```
python
```

```
# Check using color name
grader.check_plot_color('red')
grader.check_plot_color('blue')

# Check using short code
grader.check_plot_color('r')

# Check using hex code
grader.check_plot_color('#FF0000')
```

get_plot_data(line_index=0, fig_num=1)

Get the X and Y data from a plotted line.

Parameters:

- line_index (int): Which line to get data from (default: 0)
- fig_num (int): Figure number (default: 1)

Returns: Dictionary with plot data or None if not found

Dictionary Keys:

- x: X-axis data (numpy array)
- y: Y-axis data (numpy array)
- color: Line color
- linestyle: Line style ('-', '--', etc.)
- linewidth: Line width
- label: Line label

Example:

```python
# Get data from the first line
data = grader.get_plot_data(line_index=0)

if data:
    print(f"X data: {data['x']}")
    print(f"Y data: {data['y']}")
    print(f"Color: {data['color']}")
    print(f"Label: {data['label']}")
```

Code Pattern Checking

`check_code_contains(phrase, case_sensitive=True)`

Check if a specific phrase/pattern appears in the code.

Parameters:

- `phrase` (str): The phrase to search for
- `case_sensitive` (bool): Whether search should be case sensitive (default: True)

Returns: `bool` - `True` if phrase is found

Example:

```python
```

```python
# Check for string formatting
grader.check_code_contains('%0.3f')
grader.check_code_contains(':.2f')

# Check for specific code patterns
grader.check_code_contains('for i in range')
grader.check_code_contains('if __name__ == "__main__"')

# Case insensitive search
grader.check_code_contains('HELLO', case_sensitive=False)
```

`check_operator_used(operator)`

Check if a specific operator is used in the code (AST-based, more accurate).

Parameters:

- `operator` (str): The operator to check for

Supported Operators:

- Augmented assignment: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=`, `^=`, `>>=`, `<<=`
- Arithmetic: `+`, `-`, `*`, `/`, `//`, `%`, `**`
- Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
- Logical: `and`, `or`, `not`
- Membership: `in`, `not in`
- Identity: `is`, `is not`
- Bitwise: `&`, `|`, `^`, `>>`, `<<`

Returns: `bool` - `True` if operator is used

Example:

```python
# Check for augmented assignment
grader.check_operator_used('+=')
grader.check_operator_used('**=')

# Check for comparison
grader.check_operator_used('==')
grader.check_operator_used('!=')

# Check for arithmetic
grader.check_operator_used('//')
grader.check_operator_used('**')

# Check for logical operators
grader.check_operator_used('and')
grader.check_operator_used('or')
```

Control Structure Checking

check_for_loop_used()

Check if a for loop is used in the code.

Returns: bool - True if for loop is found

Example:

```python
```

```
grader.check_for_loop_used()
```

check_while_loop_used()

Check if a while loop is used in the code.

Returns: bool - True if while loop is found

Example:

```
python

grader.check_while_loop_used()
```

check_if_statement_used()

Check if an if statement is used in the code.

Returns: bool - True if if statement is found

Example:

```
python

grader.check_if_statement_used()
```

Loop Iteration Counting

count_loop_iterations(loop_variable, expected_count=None, tolerance=0)

Count loop iterations by checking a counter variable created by the student.

Parameters:

- `loop_variable` (str): Name of the counter variable
- `expected_count` (int, optional): Expected number of iterations
- `tolerance` (int): Allowed difference from expected (default: 0)

Returns: `int` or `None` - The counter value

Student Code Requirement:

```python
# Student must create a counter variable
count = 0
for i in range(10):
    count += 1
```

Example:

```python
# Just get the count
iterations = grader.count_loop_iterations('count')
print(f"Loop ran {iterations} times")

# Check against expected value
grader.count_loop_iterations('count', expected_count=10)

# Allow some tolerance
grader.count_loop_iterations('count', expected_count=100, tolerance=5)
```

Results and Reporting

`get_summary()`

Get a summary of all test results.

Returns: Dictionary with:

- `total_tests` (int): Total number of tests run
- `passed` (int): Number of tests passed
- `failed` (int): Number of tests failed
- `success_rate` (float): Percentage of tests passed
- `score` (str): Score in format "passed/total"
- `results` (list): List of all test results

Example:

```python
summary = grader.get_summary()
print(f"Score: {summary['score']}")
print(f"Success Rate: {summary['success_rate']:.1f}%")

# Access individual results
for result in summary['results']:
    status = "PASS" if result['passed'] else "FAIL"
    print(f"{status}: {result['message']}")
```

`print_summary()`

Print a formatted summary of all test results to console.

Example:

```
python

grader.print_summary()
```

Output:

```
============================================================
AUTOGRADER SUMMARY
============================================================
Score: 13/15
Total Tests: 15
Passed: 13
Failed: 2
Success Rate: 86.7%
============================================================
```

Complete Examples

Example 1: Basic Variable and Function Checking

```
python
```

```
from autograder import AutoGrader

grader = AutoGrader("student_hw1.py")
grader.execute_script()

# Check variables
grader.check_variable_value('total', 100)
grader.check_variable_value('average', 20.5, tolerance=0.1)
grader.check_variable_type('name', str)

# Check functions
grader.check_function_exists('calculate_sum')
grader.check_function_called('print')

grader.print_summary()
```

Example 2: List and Array Checking

```
python
```

```
from autograder import AutoGrader
import numpy as np

grader = AutoGrader("student_lists.py")
grader.execute_script()

# Check list with exact order
grader.check_list_equals('numbers', [1, 2, 3, 4, 5])

# Check list without order
grader.check_list_equals('unsorted', [5, 3, 1, 4, 2], order_matters=False)

# Check NumPy array
grader.check_array_equals('data_array', [1.5, 2.5, 3.5], tolerance=0.01)

# Check 2D array
grader.check_array_equals('matrix', [[1, 2], [3, 4]])

grader.print_summary()
```

Example 3: Solution File Comparison

python

```
from autograder import AutoGrader

grader = AutoGrader("student_hw3.py")
grader.execute_script()

# Compare with solution file
grader.compare_with_solution(
    solution_file='solutions/hw3_solution.py',
    variables_to_compare=['mean', 'variance', 'std_dev', 'result'],
    tolerance=0.01
)

grader.print_summary()
```

Example 4: Plot Validation

python

```python
from autograder import AutoGrader

grader = AutoGrader("student_plot.py")
grader.execute_script()

# Check plot exists
grader.check_plot_created()

# Check plot properties
grader.check_plot_properties(
    title='Data Visualization',
    xlabel='X Values',
    ylabel='Y Values',
    has_legend=True,
    has_grid=True
)

# Check plot data length
grader.check_plot_data_length(min_length=50)

# Check plot function
import numpy as np
grader.check_plot_function(lambda x: 2*x + 1)

# Check plot color
grader.check_plot_color('blue')

grader.print_summary()
```

Example 5: Comprehensive Grading

python

```
from autograder import AutoGrader
import numpy as np

grader = AutoGrader("student_comprehensive.py", timeout=15)
grader.execute_script()

# Variable checks
grader.check_variable_value('n_samples', 1000)
grader.check_variable_type('data', list)
grader.check_list_equals('sorted_data', [1, 2, 3, 4, 5])

# Function checks
grader.check_function_exists('process_data')
grader.check_function_called('np.mean')
grader.check_function_called('plt.hist')

# Control structure checks
grader.check_for_loop_used()
grader.check_if_statement_used()
grader.check_operator_used('+=')

# Code pattern checks
grader.check_code_contains('import numpy as np')

# Solution comparison
grader.compare_with_solution(
    'solutions/final.py',
    ['result', 'mean', 'std'],
    tolerance=0.01
)

# Plot validation
grader.check_plot_created()
```

```
grader.check_plot_properties(title='Histogram', has_legend=False)

# Print final summary with score
grader.print_summary()

# Get programmatic access to results
summary = grader.get_summary()
if summary['success_rate'] >= 80:
    print(f"✓ Student passed with score: {summary['score']}")
else:
    print(f"✗ Student needs revision. Score: {summary['score']}")
```

Tips and Best Practices

1. **Always execute the script first** before checking variables or testing functions
2. **Use appropriate tolerance** for floating-point comparisons
3. **Check existence before testing** - verify functions exist before calling `test_function()`
4. **Use `order_matters=False`** for lists when order doesn't matter
5. **Store solution files** in a `solutions/` folder for organization
6. **Test solution files independently** before using them for comparison
7. **Use reasonable tolerances** (0.0 for exact, 0.01-0.1 for calculations)
8. **Review the summary** at the end to get overall score and success rate

---

Supported Test Types (for Excel Configuration)

When using the GUI application, these test types are available:

Test Type	Description
`variable_value`	Check variable value
`variable_type`	Check variable type
`list_equals`	Check list equality (with/without order)
`array_equals`	Check NumPy array equality
`compare_solution`	Compare variables with solution file
`function_exists`	Check if function is defined
`function_called`	Check if function is called
`for_loop_used`	Check for 'for' loop
`while_loop_used`	Check for 'while' loop
`if_statement_used`	Check for 'if' statement
`operator_used`	Check if operator is used
`code_contains`	Check if code contains phrase
`plot_created`	Check if plot was created
`plot_properties`	Check plot title, labels, legend, grid
`plot_data_length`	Check number of data points in plot
`loop_iterations`	Check loop iteration count

Error Handling

The AutoGrader handles common errors gracefully:

- Missing files
- Syntax errors in student code
- Runtime errors during execution

- Timeout for infinite loops
- Missing variables or functions
- Type mismatches

All errors are logged and reported in the test results.

Limitations

1. **Timeout mechanism**: Uses threading which cannot forcefully kill infinite loops, but will prevent the grader from hanging
2. **Security**: Uses restricted builtins but is not a complete sandbox. For production use with untrusted code, consider Docker containers
3. **Platform**: Cross-platform (Windows, Mac, Linux)

Support

For comprehensive GUI application documentation, see the GUI Setup Instructions.

For building executables, see the Executable Build Guide.

Version: 2.0
Last Updated: 2025