Project #4: Brewin# Interpreter
CS131 Fall 2024
Due date: 12/4 11:59pm (Wednesday)

Important Info:

- Expect project #4 to take 20-30 hours of time. In our opinion, it is *at least* as difficult as project #3! **Start early!!!**

- ***This project builds on project 2, NOT project 3!!!***

- If you use our solution, ***do not fork our GitHub repo for project #2***, since it will force your forked repo to be public. If/when folks cheat off your publicly-posted code, you'll have to explain to the Dean's office why you're not guilty of cheating. Instead, copy-paste the solution files into your existing environment!

- Make sure to initialize a new git repo before starting! Just like project #2 and 3, we *might* ask you to submit your git history! More info about this here

- Your gradescope score is your **final score**, and you may submit to gradescope as many times as you'd like

DON'T WAIT UNTIL THE LAST MINUTE TO DO THIS PROJECT!

Table of Contents

Introduction

The Brewin standards body (aka Carey) has met and has decided to abandon Brewin++ and develop new improvements to the original Brewin language. They've named this new language variation Brewin#. In this project, you will be updating your Brewin interpreter so it supports these new Brewin# features. As before, you'll be implementing your interpreter in Python, and you may solve this problem using either **your original Project #2 solution**, or by modifying the **Project #2 solution that we provide**.

NOTE: Project #4 is based on Project #2 and not on Project #3!

Once you successfully complete this project, your interpreter should be able to run syntactically-correct Brewin# programs. **In particular, it should run syntactically-correct Brewin (Project #2) programs that do not generate errors and do not use strict evaluation of logical operators.**

So what new language features were added to Brewin#? Here's the list:

Need Semantics and Lazy Evaluation

Brewin# implements *need semantics* with *lazy evaluation*, meaning that expressions are evaluated only when their values are needed and no sooner. Moreover, once an expression has been evaluated its result is cached so it need not be reevaluated a second time.

Here is a simple program showing *need semantics*.

```
func main() {
  var result;
  result = f(3) + 10;
  print("done with call!");
  print(result);  /* evaluation of result happens here */
  print("about to print result again");
  print(result);
}

func f(x) {
  print("f is running");
  var y;
  y = 2 * x;
  return y;
}
```

The above program would print:

done with call!
f is running
16
about to print result again
16

Explanation:

- In `main`, the expression `f(3) + 10` is assigned to `result`, but under need semantics, the call to `f(3)` is delayed until `result` is actually used.
- `print("done with call!")` is executed immediately, so `"done with call!"` is printed first.
- When the first `print(result)` is reached, the value of `result` is required, so `f(3)` is evaluated.
  - The function `f(3)` prints `"f is running"`, computes `2 * 3 = 6`, and returns `6`. Then, `result = 6 + 10 = 16`.
  - `16` is printed for `result`.
- When the second `print(result)`, the language simply reuses the already computed value of `result`, which is `16`.

Exception Handling

Brewin# now supports simple exception handling. A function can throw an exception by using the *raise* statement and specifying a string literal, expression or variable indicating the exception name. You then use a try/catch block to execute code that may raise an exception.

Here is a simple program showing exception handling:

```
func foo() {
  print("F1");
  raise "except1";
  print("F3");
}

func bar() {
 try {
   print("B1");
   foo();
   print("B2");
 }
 catch "except2" {
   print("B3");
 }
 print("B4");
}

func main() {
 try {
```

```
    print("M1");
    bar();
    print("M2");
  }
  catch "except1" {
    print("M3");
  }
  catch "except3" {
    print("M4");
  }
  print("M5");
}
```

The above program would print:

M1
B1
F1
M3
M5

Explanation:

- **main** prints M1 and calls bar().
- **bar** prints B1 and calls foo().
- **foo** prints F1 and raises "except1", skipping the rest of foo().
- The exception propagates up to bar(), but there's no catch statement for "except1", so bar() doesn't complete and the exception goes back to main().
- **main** catches "except1", prints M3, and continues with M5.

Short Circuiting

Brewin# now supports short circuiting for the && and || operators.

Here is an example:

```
func foo() {
  print("foo");
```

```
 return true;
}

func bar() {
 print("bar");
 return false;
}

func main() {
  print(foo() || bar() || foo() || bar());
  print("done");
}
```

`The above program would print:`

foo
true
done

Explanation:

- The first `foo()` returns `true`, so the remaining expressions (`bar() || foo() || bar()`) are skipped due to short-circuiting.
- Only `"foo"` is printed while evaluating the logical expression.
- The first call to `print` prints `"true"`, and the second one prints `"done"`.

What Do You Need To Do For Project #4?

Now that you have a flavor for the new language features, let's dive into the details.

For this project, you will create a new class called `Interpreter` within a file called `interpreterv4.py` and derive it from our `InterpreterBase` class (found in our provided `intbase.py`). As with Project #2, your `Interpreter` class **MUST** implement at least the constructor and the `run()` method that is used to interpret a Brewin# program, so we can test your interpreter. You may add any other public or private members that you like to your `Interpreter` class. You may also create other modules (e.g., `variable.py`) and leverage them in your solution.

Brewin# Language Specification

The following sections provide detailed requirements for the Brewin# language so you can implement your interpreter correctly.

Need Semantics and Lazy Evaluation

In the context of lazy evaluation in this specification, we use ***expression*** to refer to expression nodes (which are binary or unary operations and function calls), and standalone variable nodes (which appear on the right-hand side of assignments, in return and raise statements, as function parameters or as for/in conditionals).

We also emphasize the distinction between *standalone* function calls represented by statement nodes such as

```
print("x");
foo();
inputi("Enter a number");
```

and function calls within expressions such as

```
a = foo();
n = inputi("Enter a number");
return (print(x) == nil);
```

In Brewin#, all expressions (as defined above) are evaluated using lazy evaluation with need semantics. This means that expressions are (eagerly) evaluated only when their values are actually needed, such as

1) when they are used by standalone `print` or `input` calls,
2) in an if/for conditional,
3) in a raise statement.

Moreover, once an expression has been evaluated, its result is cached so it does not need to be re-evaluated if used again.

Example:

```
func main() {
  var result;
  result = f(3) + 10;    /* f(3) + 10 evaluation is deferred */
```

```
  if (result > 5) {      /* result is evaluated here */
    ...
  }

  var i;
  for (i = 0; i < 10; i = i + 1) {  /* i is evaluated during i < 10 */
    ...
  }

  result = "except" + "9"  /* concatenation operation is deferred */

  ...
  raise result;            /* result is evaluated here */
}

func f(x) {
  ...
}
```

Errors and Exceptions During Lazy Evaluation

Errors, including type errors and name errors, **do not occur unless and until an expression is eagerly evaluated**. This means that during lazy evaluation, errors will not be generated. The evaluation of expressions is deferred until their values are needed, and any errors will be raised at that time.

Example:

```
func faultyFunction() {
  print(undefinedVar); /* Name error occurs here when evaluated */
}

func main() {
  var result;
  result = faultyFunction();
  print("Assigned result!");

  print(result);      /* Error will occur when result is evaluated */
}
```

When running this program, the output would be:

```
Assigned result!
```

And then a `NAME_ERROR` would be raised when `print(result);` is called, because `faultyFunction()` tries to use an undefined variable when evaluated.

The same holds true for exceptions. **Exceptions**, including raised exceptions and divide by zero exceptions, **do not occur unless and until the code that generates the exception is eagerly evaluated**. This means that during lazy evaluation, exceptions will not be generated. The evaluation of expressions is deferred until their values are needed, and any errors will be raised at that time.

Example:

```
func functionThatRaises() {
  raise "some_exception";  /* Exception occurs here when func is called */
  return 0;
}

func main() {
  var result;
  result = functionThatRaises();
  print("Assigned result!");
  /* Exception will occur when result is evaluated */
  print(result, " was what we got!");
}
```

Explanation:

- `result = functionThatRaises()` does not immediately raise an exception because the function is not evaluated yet.
- The program prints `"Assigned result!"` since the exception is deferred.
- The exception only occurs when `result` is evaluated in `print(result)`, causing the program to terminate before printing " was what we got!".

Requirements

- **Lazy Evaluation of Expressions**:
    - All expressions are evaluated lazily unless they are in a context that requires eager evaluation (i.e., if/for conditionals, arguments of *standalone* calls to `inputi/inputs/print`, `raise`).
    - This includes (expression node) function calls, arithmetic expressions, logical expressions, right-hand side variables, etc.

- All parameters passed to functions (both standalone function calls *other than* `print` and `input`, and those used within expressions) are evaluated lazily as well (e.g., not prior to the function call, but only when the value is needed - see below)
- Function calls, including calls to inputi() and inputs(), within expressions are also evaluated lazily unless they are in an eager evaluation context.
- The expressions in `return` statements are evaluated lazily.
- Standalone function calls are executed immediately. For instance, function call

  foo(x+3);

  will evaluate x+3 lazily and start running foo but in assignment

  y = foo(x+3);

  the entire right-hand side is evaluated lazily, so the execution of foo is deferred to the point when y is evaluated eagerly.

- Function calls can have side effects, so the ordering of their evaluation can affect program output.
- Once an expression is evaluated eagerly, all subexpressions that make up the expression must also be evaluated eagerly.
- Later changes to variables involved in earlier expressions have no impact on evaluation of the earlier expression, e.g.:

  x = 5;
  y = x + 10;
  x = 100;
  print(y); /* still prints 15 */

- **Caching of Results**:
  - Once an expression is evaluated, its result must be cached in the variable to which it was assigned.
  - Subsequent uses of the variable must use the cached value without re-evaluating the expression.
  - In particular, if an expression which contains a lazily evaluated variable is eagerly evaluated, the result of the variable evaluation is also cached, e.g.:

    x = foo(3);
    y = x + 10;
    print(y);
    print(x); /* foo(3) is **NOT** re-evaluated */

- **Eager Evaluation in Certain Contexts**:

- **Conditional Expressions**: The **conditions** in `if` and `for` statements e.g., if (x > 5) … or for (x=0;x < 10; x = x+ 1) … must be evaluated eagerly because the control flow depends on their values.
- **Built-in Functions**: Arguments to built-in functions such as `print()`, `inputi()` and `inputs()` must be evaluated eagerly when these functions are executed as standalone function calls.
- **Raise**: The argument to the raise statement must be evaluated eagerly prior to the exception being generated.

- **Order of Evaluation**:
  - When expressions are eventually evaluated, evaluation proceeds from left to right, e.g. foo() + bar() * baz() would have foo() called first, followed by bar(), and then baz().
  - Same holds true for arguments when `print` is called: they are eagerly evaluated from left to right
  - Side effects in expressions (e.g., function calls that print) occur at the time of evaluation.
- **Errors and Exceptions:**
  - Errors will be generated only during eager evaluation of an expression, e.g.:

```
func main() {
  var x;
  x = foo(y);
  print("OK");
  print(x);  /* NAME_ERROR due to undefined y is deferred to this line */
}
```

  - All errors outside of the lazy evaluation context should be generated immediately, e.g.:

```
func main() {
  x = foo();  /* generates NAME_ERROR immediately since x is undefined */
              /* and x is not part of expression                      */
  print("OK");
  print(x);
}
```

  - Exceptions will only be generated during eager evaluation of an expression.

Exception Handling

Brewin# introduces simple exception handling mechanisms to allow programs to handle errors gracefully.

Raise Statement

raise expression/variable/value;

Examples:

```
raise "foo";
x = "foo"+"bar";
raise x;
raise "foo" + "bar";
```

Try/Catch Blocks

```
try {
  /* statements that may raise exceptions */
}
catch "exception_type_1" {
  /* handler for exception named "exception_type_1" */
}
…
catch "exception_type_n" {
  /* handler for exception named "exception_type_n" */
}
```

Multiple `catch` clauses can be used to handle different exception types. You may assume all `try` clauses are accompanied by at least one `catch` clause, and all `catch` clauses handle *distinct* exception types. A `try` clause with no `catch` clauses is a syntax error and therefore will not be tested.

Just like `if` and `for` scopes, `try/catch` scopes must support variable ***shadowing*** rules as outlined in project #2

Examples:

```
func foo() {
  try {
    raise "z";
  }
```

```
  catch "x" {
    print("x");
  }
  catch "y" {
    print("y");
  }
  catch "z" {
    print("z");
    raise "a";
  }
  print("q");
}

func main() {
  try {
    foo();
    print("b");
  }
  catch "a" {
    print("a");
  }
}
```

Expected Output:

```
z
a
```

Explanation:

- `foo()` raises an exception of type `"z"`.
- The exception is caught by the `catch "z"` clause within `foo()`, printing `"z"`.
- Inside the `catch "z"` block, another exception `"a"` is raised.
- The exception `"a"` is not caught within `foo()`, so it propagates back to `main()`.
- In `main()`, the `catch "a"` clause catches the exception and prints `"a"`.
- The program terminates after handling the exception.

Additional Example:

Combining Lazy Evaluation with Exception Handling:

```
func error_function() {
  raise "error";
  return 0;
}

func main() {
  var x;
  x = error_function() + 10;  // Exception occurs when x is evaluated
  print("Before x is evaluated");
  try {
    print(x);  // Evaluation of x happens here
  }
  catch "error" {
    print("Caught an error during evaluation of x");
  }
}
```

Expected Output:

```
Before x is evaluated
Caught an error during evaluation of x
```

Explanation:

- The expression `error_function() + 10` assigned to `x` is not evaluated at the time of assignment.
- The `print("Before x is evaluated")` statement executes without error.
- When `print(x)` is called, `x` is evaluated, causing `error_function()` to be called.
- `error_function()` raises an exception `"error"`.
- The exception is caught by the `catch "error"` block, and `"Caught an error during evaluation of x"` is printed.

Division by Zero Exceptions

In Brewin#, attempting to divide by zero during eager evaluation results in a `"div0"` exception being raised. This exception can be caught using a `try/catch` block.

Example:

```
func divide(a, b) {
  return a / b;
}

func main() {
  try {
    var result;
    result = divide(10, 0);  /* evaluation deferred due to laziness */
    print("Result: ", result); /* evaluation occurs here */
  }
  catch "div0" {
    print("Caught division by zero!");
  }
}
```

Expected Output:

```
Caught division by zero!
```

Explanation:

- At the time the `result` variable is printed, the `divide` function is called and attempts to compute `10 / 0`, which is undefined.
- When the division by zero is attempted, a `"div0"` exception is raised.
- The `catch "div0"` block catches the exception and prints `"Caught division by zero!"`.
- The program continues execution after handling the exception.

Requirements

- **Raise Statement**:
  - The `raise` statement causes an exception to be thrown.
  - The exception type is specified by an expression, variable, or value that evaluates to a string.

- The expression passed to the raise statement must be evaluated eagerly before the exception is thrown.
- If the expression does not evaluate to a string, you must generate an error of type `ErrorType.TYPE_ERROR` by calling `InterpreterBase.error()`.

- **Try/Catch Blocks**:
  - The `try` block contains code that may raise exceptions.
  - One or more `catch` clauses follow the `try` block.
  - Each `catch` clause specifies the exception type it can handle ***(as a string literal)***. Passing expressions other than string literals to a catch clause is a syntax error and will not be tested.
  - If an exception is raised within the `try` block, control is transferred to the `catch` clause that exactly matches the exception type.
  - If no matching `catch` clause is found in the current try block, the exception propagates to the innermost enclosing try block, then the next innermost enclosing try block, etc., and then to the calling function.
  - In particular, if an exception is generated inside a `catch` clause, it will *not* be caught by any `catch` clause corresponding to the *same* `try` block
  - Try/catch blocks may be nested, so you may have arbitrarily many try/catch blocks nested inside enclosing try/catch blocks.
  - After the `try` block finishes without raising an exception or one of the `catch` clauses is finished running, the control is transferred to the next line after all the `catch` clauses.
- **Exception Propagation**:
  - Exceptions propagate up the call stack until they are caught.
  - If an exception is not caught anywhere, you must generate an error of type `ErrorType.FAULT_ERROR` by calling `InterpreterBase.error()`.
- **Exception Types**:
  - Exception types are strings.
  - Exception handling uses string equality to match exceptions. The raised exception string must match exactly with a string in a catch clause to be caught.
- **Built-in Exceptions**:
  - Your code must generate division by zero exceptions with an exception type of "div0" any time an eager evaluation results in a division by zero
- **Handling Exceptions in Expressions:**
  - If an operand within an expression raises an exception (e.g., `f() + g()`, where `f()` raises an exception), evaluation of the expression immediately ceases and control is transferred to the appropriate `catch` block. Further evaluation of the expression (e.g., evaluation of `g()`) is aborted after the exception is raised.
  - If an expression raises an exception, you can assume that the programs we test you on will not attempt to evaluate that expression more than one time.
- **Handling Exceptions in Statements:**

    - If an expression raises an exception during eager evaluation in an if statement condition, for statement condition, print or inputi/inputs argument, all further processing of that statement is aborted and the exception is propagated to the nearest catch.
    - If, while evaluating the operands to a print statement, any of the operands to the results in an exception then the entire print statement will be aborted and nothing must be printed out.
    - We will never test a situation where an expression to a raise operation itself throws an exception, e.g.:

      raise this_func_raises_its_own_exception(); /* will not be tested */

- **Variables in Catch Blocks:**
    - Variables defined within the `try` block are not accessible in the corresponding `catch` clauses.
    - When a try block exits either normally or due to an exception, all local variables defined in the `try` block go out of scope and their lifetime ends.
    - Variables defined before the `try` block are accessible within both the `try` and `catch` blocks.

Short Circuiting for AND and OR Operators

Brewin# now supports short-circuit evaluation for the logical operators && (AND) and `||` (OR).

Behavior:

- **Logical AND (&&)**:
    - If the first operand evaluates to `false`, the second operand is **not** evaluated, and the result is `false`.
    - If the first operand evaluates to `true`, the result depends on the evaluation of the second operand.
- **Logical OR (`||`)**:
    - If the first operand evaluates to `true`, the second operand is **not** evaluated, and the result is `true`.
    - If the first operand evaluates to `false`, the result depends on the evaluation of the second operand.

Example:

```
func t() {
```

```
 print("t");
 return true;
}

func f() {
 print("f");
 return false;
}

func main() {
  print(t() && f());
  print("---");
  print(f() && t());
}
```

Expected Output:

```
t
f
false
---
f
false
```

Explanation:

- In the expression `t() && f()`:
  - `t()` is called first, printing `"t"` and returning `true`.
  - Since the first operand is `true`, the second operand `f()` is evaluated.
  - `f()` is called, printing `"f"` and returning `false`.
  - The result of the `&&` operation is `false`, which is printed.
- In the expression `f() && t()`:
  - `f()` is called first, printing `"f"` and returning `false`.
  - Since the first operand is `false`, the second operand `t()` is not evaluated due to short-circuiting.
  - The result of the `&&` operation is `false`, which is printed.

Requirements

- **Short-Circuit Evaluation**:
  - Implement short-circuiting behavior for the `&&` and `||` operators.
  - Do not evaluate the second operand if the result can be determined from the first operand.
  - Ensure that side effects (e.g., function calls, print statements) in the second operand do not occur if short-circuiting occurs.
  - Ensure that exceptions and errors in the second operand do not get processed if short-circuiting occurs.

Abstract Syntax Tree Specification

You need to handle additional AST node types to support the new features in Brewin#.

New AST Nodes:

- Try Node
- Catch Node
- Raise Node

Try Node

A *Statement* node representing a `try` statement, which includes a block of statements to execute and a list of catch clauses.

Fields:

- `self.elem_type`: `'try'`
- `self.dict`: Contains two keys:
  - `'statements'`: A list of Statement nodes representing the statements inside the `try` block.
  - `'catchers'`: A list of Catch nodes representing each `catch` clause associated with the `try` block.

Catch Node

A Catch node represents a `catch` clause, which specifies an exception type to catch and a block of statements to execute when that exception is caught.

Fields:

- `self.elem_type`: `'catch'`
- `self.dict`: Contains two keys:
    - `'exception_type'`: A string representing the exception type that this `catch` clause handles (e.g., `"except1"`).
    - `'statements'`: A list of Statement nodes representing the statements inside the `catch` block.

Raise Node

A *Statement* node representing a `raise` statement, which raises an exception.

Fields:

- `self.elem_type`: `'raise'`
- `self.dict`: Contains one key:
    - `'exception_type'`: An Expression node, Variable node, or Value node representing the exception type to raise.

Things We Will and Won't Test You On

You may assume the following when building your interpreter:

- **WE WILL NOT TEST YOUR INTERPRETER ON SYNTAX ERRORS OF ANY TYPE**
    - All programs we present to your interpreter will be syntactically well-formed.
    - There will be no mismatched parentheses, mismatched quotes, or missing syntactic elements.
- **WE WILL TEST YOUR INTERPRETER ON ONLY THOSE SEMANTIC AND RUN-TIME ERRORS EXPLICITLY SPECIFIED IN THIS SPEC**
    - You must handle the specified errors using `InterpreterBase.error()`.
    - Examples include:
        - Using `raise` with a non-string exception type.
        - Exceptions not caught anywhere.
        - Type or name errors in expressions when evaluating operands.
- **WE WILL NOT TEST YOUR INTERPRETER ON EFFICIENCY, EXCEPT: YOUR INTERPRETER NEEDS TO COMPLETE EACH TEST CASE WITHIN 5 SECONDS**
    - You don't need to optimize for efficiency, but your interpreter must avoid infinite loops or extremely slow execution.

- **WHEN WE SAY YOUR INTERPRETER MAY HAVE "UNDEFINED BEHAVIOR" IN A PARTICULAR CIRCUMSTANCE, WE MEAN IT CAN DO ANYTHING YOU LIKE AND YOU WON'T LOSE POINTS**
    - For cases where the spec states that your program may have undefined behavior, your interpreter can behave in any way, including crashing or producing unexpected results.

Coding Requirements

You **MUST** adhere to all of the coding requirements stated in Project #2, and:

- You must name your interpreter source file `interpreterv4.py`.
- You may submit as many other supporting Python modules as you like (e.g., `statement.py`, `variable.py`, etc.) which are used by your `interpreterv4.py` file.
- Try to write self-documenting code with descriptive function and variable names and use idiomatic Python code.
- You **MUST NOT** modify our `intbase.py`, `brewparse.py`, or `brewlex.py` files since you will **NOT** be turning these files in. If your code depends upon modified versions of these files, this will result in a grade of zero on this project.
- We **strongly** encourage you to keep an incremental git history documenting your progress on the project — though it is not required. In case your code resembles another submission, this is a great way to prove you did the work yourself!
    - In general, it's a good idea to commit when you add a feature, fix a bug, or refactor some code. Here's an example of an ideal commit history. Here's an example of a commit history that is lacking.

Deliverables

For this project, you will turn in at least two files via GradeScope:

- Your `interpreterv4.py` source file.
- A `readme.txt` indicating any known issues/bugs in your program (or, "all good!").
- Other Python source modules that you created to support your `interpreterv4.py` module (e.g., `variable.py`, `type_module.py`).

You **MUST NOT** submit `intbase.py`, `brewparse.py`, or `brewlex.py`; we will provide our own. You must not submit a `.zip` file. On Gradescope, you can submit any number of source files when uploading the assignment; assume (for import purposes) that they all get placed into one folder together.

We will be grading your solution on Python 3.11. Do not use any external libraries that are not in the Python standard library.

Whatever you do, make sure to turn in a Python script that is capable of loading and running, even if it doesn't fully implement all of the language's features. We will test your code against dozens of test cases, so you can get substantial credit even if you don't implement the full language specification.

The TAs have created a template GitHub repository that contains `intbase.py` (and a parser `brewparse.py`) as well as a brief description of what the deliverables should look like.

Grading

Your score will be determined entirely based on your interpreter's ability to run Brewin programs correctly (however you get karma points for good programming style). A program that doesn't run with our test automation framework will receive a score of 0%.

The autograder we are using, as well as a subset of the test cases, is publicly available on GitHub. Other than additional test cases, the autograder is exactly what we deploy to Gradescope. Students are encouraged to use the autograder framework and provided test cases to build their solutions. Students are also STRONGLY encouraged to come up with their own test cases to proactively test their interpreter.

We strongly encourage you to write your own test cases. The TAs have developed a tool called barista that lets you test any Brewin code and provide the canonical response. In discussion, TAs will discuss how to use our test infrastructure and write your own test cases.

Your score on this project will be the score associated with the final project submission you make to GradeScope. In other words, the grade you see on GradeScope is your final grade![1]

Academic Integrity

The following activities are NOT allowed - all of the following will be considered **cheating**:

- Publishing your source code on an open GitHub repo where it might be copied (you MAY post your source code on a public repo after the end of the quarter)

[1] As long as you submit on time.

    - Warning, if you clone our public GitHub repo, it will force the clone to also be public. So create your own private repo to avoid having your code from being used by another student!
- Leveraging ANY source code from another student who is NOW or has PREVIOUSLY been in CS131, IN ANY FORM
- Sharing of project source code with other students
- Helping other students debug their source code
- Hacking into our automated testing or Barista systems, including, but not limited to:
    - Connecting to the Internet (i.e., from your project source code)
    - Accessing the file system of our automated test framework (i.e., from your project source code)
    - Any attempts to exfiltrate private information from our testing or Barista servers, including but not limited to our private test cases
    - Any attempts to disable or modify any data or programs on our testing or Barista servers
    - Any attempts to cause our framework to give you a different score/grade then your solution would otherwise earn
- Leveraging source code from the Internet (including ChatGPT) without a citation comment in your source code
- Collaborating with another student to co-develop your project

The following activities ARE explicitly allowed:

- Discussing general concepts (e.g., algorithms, data structures, class design) with classmates or TAs that do not include sharing of source code
- Sharing test cases with classmates, including sharing the source code for test cases
- Including UP TO 50 TOTAL LINES OF CODE across your entire project from the Internet in your project, so long as:
    - It was not written by a current or former student of CS131
    - You include a citation in your comments:

```
# Citation: The following code was found on www.somesite.com/foo/bar
... copied code here
# End of copied code
```

  Note: You may have a TOTAL of 50 lines of code copied from the Internet, not multiple 50-line snippets of code!

- Using ChatGPT, CoPilot or similar code generation tools to generate snippets of code so long as you include a citation in your comments. See the syllabus for more concrete guidelines surrounding LLMS.
- Using ChatGPT, CoPilot or similar code generation tools to generate any number of test cases, test code, etc.