ICS 33 Spring 2025
Project 3: *Why Not Smile?*

Due date and time: *Monday, May 19, 11:59pm*
Final late work deadline: *Saturday, May 24, 2:59am*

Git repository: *https://ics.uci.edu/~thornton/ics33/ProjectGuide/Project3/Project3.git*

Introduction

When I was a young kid, one of my teachers introduced me to a computer for the first time; it was a state of the art (in those days) personal computer called a Radio Shack TRS-80 Model I. First, I played little math games and messed around with other new-fangled educational tools from 1980; the state of the art wasn't much then, but it was fun and new, and felt alive with possibility.

Booting up a TRS-80 took the user directly into the equivalent of a Python shell; you could load programs from external storage like floppy disks or cassette tapes, but the computer's default mode was an environment for writing programs. My teacher asked me if I wanted to learn how to write my own programs, which I thought sounded like a great idea, though I had no idea how to do it. So, I opened up a book of his about the TRS-80's primary programming language, which was called BASIC, which was a good teaching and learning tool for its day: versatile and easy to start with, much like Python is today. I typed in a short program that asked a user for a number of hits and a number of at-bats and printed out a batting average (foreshadowing my later interest in baseball, though I didn't know what it meant at the time). I ran the program, tried it out, and I was mesmerized; the computer did exactly what I asked it to, exactly the way I asked it to. I was hooked. Over forty years later, I still am.

A natural progression of one's curiosity about programming revolves around the question of how to implement one's own programming language. Where do they come from? How are they built? While we won't be able to tackle these questions in too much depth — there are at least three different courses in our undergraduate curriculum that cover aspects of this — this project will ask you to begin exploring them. For that purpose, I've designed a considerably limited (and somewhat different) version of BASIC called Grin, which supports a small handful of statements. You'll be building a Grin *interpreter*, a program written in Python that takes a Grin program as its input, executes the Grin program, then shows its output. (This may sound a little mind-bending, but it's not as crazy as it sounds. The Python interpreter you've been using was most likely written in a language other than Python; the most popular one is written in a language called C.)

In the process of building your interpreter, you'll gain experience in a few areas that will stretch your abilities:

- Continuing to develop your understanding of object-oriented design, as you'll be on the lookout for concepts in the program that would best be represented as *classes*.
- Using *inheritance*, which is Python's mechanism for defining new classes in terms of existing ones, an important technique when you have many classes that share at least some of the same behavior.
- Writing *unit tests* incrementally that cover as much as of your program as it practical, so that you can verify that parts of your program work as you expect before you build larger parts on top of them.

The Grin language

The precise requirements for your interpreter are discussed later in this write-up, but we'll first need to agree on the definition of the Grin language that your interpreter will implement. Grin is a programming language, though its design is quite different from Python's, so we'll first need to acquaint ourselves with how it works. Given a Grin program, you'll need to know, first and foremost, what its output is meant to be.

A Grin program is a sequence of *statements*, one per line. Here's an example of a Grin program:

```
LET MESSAGE "Hello Boo!"
PRINT MESSAGE
.
```

Each line contains exactly one statement (i.e., there can be no blank lines). Grin assigns a *line number* to each of the statements, where the first statement in the program is numbered 1, the second statement is numbered 2, and so on. There is no predefined limit on the number of statements in a Grin program. Execution of a Grin program always begins at line number 1. The last line contains only a dot (.) and nothing else, as a way to mark that the program has ended; it's not a statement, but any subsequent lines of text in the Grin program after that end-of-program marker are ignored.

The program above consists of two statements. The first one stores the text `Hello Boo!` into a variable named `MESSAGE`, then the second one prints the value of that same variable. The output of the program is what you'd expect, given that description.

```
Hello Boo!
```

Lexical rules

Like most programming languages (including Python), a Grin program is made up of a sequence of *lexemes*, which is a fancy-sounding term for a sequence of characters that combine together with a single meaning and comprise one of the indivisible "atoms" in the language, similar to the role that words play in sentences written in natural languages like English. Programming languages that are written textually generally define a set of *lexical rules* that specify which lexemes are valid and how to derive a meaning for each of them; Grin is no different, in that respect, so we'll need to start our journey with Grin by acquainting ourselves with those rules.

Grin programs are made up of the following kinds of lexemes.

- *Integer literals*, which are sequences of one or more digits (0-9), optionally preceded by a minus sign `-`. Their meaning is their corresponding integer value.
- *Floating-point literals*, which are integer literals that are immediately followed by a dot `.` and, optionally, one or more digits (0-9). Their meaning is the corresponding floating-point value.
- *String literals*, which are sequences of zero or more characters, both preceded and followed by one double quote character `"`. Their meaning is the sequence of characters contained between the double quotes, but not including the double quotes.
  - There are no special escape sequences such as `\n` that you'd find in Python, which means that there are two kinds of characters that cannot appear in string literals: newlines and double quotes.

- *Identifiers*, which are used to specify the names used to describe things like variables and labels. Identifiers begin with a letter, optionally followed by a sequence of letters and digits. Identifiers in Grin are *case-sensitive*, which means that `BOO`, `Boo`, and `boo` are each considered to be different from the others.
- *Keywords*, which are sequences of zero or more characters that have a special meaning and, thus, can never be used as identifiers. The following are keywords in Grin: `ADD`, `DIV`, `END`, `GOSUB`, `GOTO`, `IF`, `INNUM`, `INSTR`, `LET`, `MULT`, `PRINT`, `RETURN`, `SUB`.
- *Comparison operators*, which can be used in some statements to compare two values. There are six comparison operators: `=`, `<>`, `<`, `<=`, `>`, and `>=`.
- *Label markers*, which are colon characters (i.e., `:`) that are used to specify the existence of a label on a line.
- *End-of-program markers*, which are dot characters (i.e., `.`) that are used to mark the end of a program.

Some examples of Grin lexemes and their meanings follow.

```
0                    # Integer literal (zero)
13                   # Integer literal (positive)
-18                  # Integer literal (negative)
0.0                  # Floating-point literal (zero)
11.75                # Floating-point literal (positive)
-3.0                 # Floating-point literal (negative)
""                   # String literal (an empty one)
"Boo!"               # String literal (containing four characters)
A                    # Identifier
BOO                  # Identifier
THIS1ISTHELAST1      # Identifier
IF                   # Keyword
GOTO                 # Keyword
=                    # Comparison operator
>=                   # Comparison operator
```

```
    :                    # Label marker
    .                    # End-of-program marker
```

Labels

Any statement in a Grin program can begin with a *label*, which is a name that can be used to refer to that statement elsewhere in the program without having to rely on knowing its line number. Labels appear at the beginning of a line, and are made up of an identifier followed by a colon.

```
        LET A 3
        PRINT A
        GOSUB "CHUNK"
        PRINT A
        PRINT B
        GOTO "FINAL"
CHUNK:  LET A 4
        LET B 6
        RETURN
FINAL:  PRINT A
        .
```

In the program above, two statements have labels on them: `LET A 4` is labeled as `CHUNK` and the last statement is labeled as `FINAL`.

Spacing

One of the features of Python's syntax is that the way you space your program — indention, empty lines, and so on — has an effect on your program's meaning. Grin, in that sense, is different. Grin programs cannot have blank lines in them, each statement must be on its own line, and at least one space is required to separate lexemes that would otherwise be combined, but the specific amount and placement of blank space between the lexemes on each line is otherwise irrelevant. So, the following program is legal and equivalent Grin to the previous one shown, though obviously there's a lot to be said for using spacing to make a program's meaning more obvious to a human reader.

```
          LET        A    3
 PRINT        A
    GOSUB    "CHUNK"
      PRINT    A
PRINT   B
   GOTO          "FINAL"
       CHUNK   :  LET A 4
LET   B                            6
            RETURN
FINAL:     PRINT    A
  .
```

Variables

A Grin program can utilize *variables* to store values that can be accessed again (or modified) later. Each variable is named by an identifier. Variables do not need to have values assigned to them before they are used, and any variable that is used before it is assigned has the integer value 0.

The primary way to change the value of a variable is with a `LET` statement. A `LET` statement changes the value of one variable, by either assigning it a literal value or the value of another variable.

- `LET A 3` — changes the value of the variable `A` to the integer `3`
- `LET NAME "Boo"` — changes the value of the variable `NAME` to the string `"Boo"`
- `LET QQQ SSS` — changes the value of the variable `QQQ` to store a copy of the value stored in the variable `SSS`

You can print the value of a variable to the output by using a `PRINT` statement. A `PRINT` statement prints the value of one variable, followed by a newline.

So, consider the following short Grin program:

```
LET NAME "Boo"
LET AGE 13.015625
PRINT NAME
PRINT AGE
.
```

Its output would be:

```
Boo
13.015625
```

The formatting rules used when printing the values of variables depend on their types.

- Integers are printed as they are in Python: An optional `-` character (for negative integers only), followed by a sequence of one or more digits without leading zeroes.
- Floating-point numbers are printed as they are in Python: An optional `-` character (for negative numbers only), followed by a sequence of one or more digits without leading zeroes, followed by a `.` character (a decimal point), followed by a sequence of one or more digits.
- Strings are printed by printing their contents (i.e., the characters within them), without double quotes around them.

Reading input

Grin includes two statements for reading input from the console:

- `INNUM`, which is used when you want to read an integer or floating-point number.
- `INSTR`, which is used when you want to read a string.

Either way, the syntax is mostly the same: We write `INNUM` or `INSTR`, followed by the name of the variable into which you want to read the input value. A short Grin program demonstrates the idea.

```
PRINT "Number:"
INNUM X
ADD X 7
PRINT X
.
```

This program prints output and also reads input, so let's imagine what that might look like when we execute it.

```
    Number:
    11
    18
```

First, the `PRINT` statement on line 1 will have printed `Number:`. Next, a line of input will have been read and treated, in this case, as the integer `11`, which will be stored in the variable named `X`. We'd then add 7 to `X`, causing its value to become the integer `18`. Finally, we'd print `X`'s value, which causes `18` to be printed.

The precise rules for `INNUM` need to be specified, though, since not all inputs are valid.

- When the user enters a sequence of digits, optionally preceded by a minus sign `-`, the value is treated as an integer. Leading or trailing whitespace is permitted.
- When the user enters a sequence of digits, optionally preceded by a minus sign `-`, followed by a dot (i.e., `.`), optionally followed by more digits, the value is treated as floating-point. Leading or trailing whitespace is permitted.

- When the user enters anything not meeting these characteristics, the program terminates with an error message.

Meanwhile, the precise rules for `INSTR` are much simpler, because not much can go wrong. We read a line of text, then store the contents of that line (without a trailing newline) into the given variable. Any line of text, including empty lines or very long lines, is permitted, so there are no error conditions to consider.

Control flow and how to alter it

A Grin program is executed one statement at a time, beginning at line number 1. Ordinarily, execution proceeds forward, so that line 1 will execute first, followed by line 2, followed by line 3, and so on. Execution continues until either an `END` statement is reached, or until execution proceeds beyond the last statement in the program.

Like most programming languages, Grin makes it possible to write programs that execute out of sequence, though the mechanisms are a bit more primitive than they are in a language like Python. A `GOTO` statement causes execution to "jump" immediately forward or backward by the given number of lines. For example, the statement `GOTO 4` jumps execution to the line number that's 4 greater than the current one. Here's an example Grin program that uses `GOTO`:

```
LET A 1
GOTO 2
LET A 2
PRINT A
.
```

In this program, line 1 is executed first, setting the variable `A`'s value to 1. Then the `GOTO` statement will immediately jump execution of the program to line 4 — the `GOTO` statement is on line 2, and two lines beyond that is line 4 — skipping the second `LET`. Line 4 prints the value of `A`, which is still 1. So, the output of the program is simply `1`.

A `GOTO` statement may jump either forward or backward, meaning that the following program is a legal Grin program. See if you can figure out what its output would be. (Remember that the value of a variable that hasn't yet been assigned with a

LET is 0.)

```
LET Z 5
GOTO 5
LET C 4
PRINT C
PRINT Z
END
PRINT C
PRINT Z
GOTO -6
.
```

Alternatively, GOTO statements can specify a string literal specifying a label instead of a line number, in which case execution jumps to the line that is marked with that label. A Grin program equivalent to the previous one, but that uses labels instead of line numbers, follows.

```
        LET Z 5
        GOTO "CZ"
CCZ:    LET C 4
        PRINT C
        PRINT Z
        END
CZ:     PRINT C
        PRINT Z
        GOTO "CCZ"
        .
```

GOTO statements can cause the program to terminate with an error message in a few circumstances.

- Jumping to a line number that's zero or negative.
- Jumping to a line number that's more than one beyond the last statement of the program. (So, in a five-statement program with a `.` on line 6, you can jump to line 6 — which will cause the program to end — but not to line 7 or greater.)
- `GOTO 0` would be guaranteed to be an infinite loop, so it is not permitted.
- Jumping to a non-existent label.

Finally, it should be noted that GOTO statements can use variables to specify their target, as long as the variable contains either an integer or a string value, in which case that value is treated the same as it would have been if it had been specified literally.

```
        LET Z 1
        LET C 11
        LET F 4
        LET B "ZC"
        GOTO F
ZC:     PRINT Z
        PRINT C
        END
CZ:     PRINT C
        PRINT Z
        GOTO B
        .
```

When the target of a `GOTO` is a variable containing something other than an integer or a string, that, too, terminates the interpreter with an error message.

Arithmetic operations

Grin provides the typical arithmetic operations that can be performed on variables: addition, subtraction, multiplication, and division. Each operation is provided as a statement that updates the value of the given variable by combining it with another value, making it equivalent to operators like `+=`, `-=`, etc., in Python. The first operand must be the name of a variable; the second can either be a literal value or the name of a variable. Here are examples of their use on integers:

```
LET A 4
ADD A 3
PRINT A
LET B 5
SUB B 3
PRINT B
LET C 6
MULT C B
PRINT C
LET D 8
DIV D 2
PRINT D
.
```

In the example above, the `ADD` statement adds `3` to the value of `A`, storing the result in `A`. So, printing `A` will display `7` on the output. The output of the entire program above is as follows.

```
7
2
12
4
```

Like Python, arithmetic operations have a different meaning when operating on different types of values. Note, though, that some of the rules in Grin are different from the ones you learned in Python. (These are among the subtleties you'll find that change from one programming language to another.)

Statement	*Type (in variable)*	*Type (in operand)*	*Result Type*	*Example*
`ADD`	Integer	Integer	Integer	11 + 7 = 18
`ADD`	Float	Float	Float	11.5 + 7.0 = 18.5
`ADD`	Integer	Float	Float	11 + 7.5 = 18.5
`ADD`	Float	Integer	Float	11.5 + 7 = 18.5
`ADD`	String	String	String	"Boo" + "lean" = "Boolean"
`SUB`	Integer	Integer	Integer	18 - 7 = 11
`SUB`	Float	Float	Float	18.5 - 7.0 = 11.5
`SUB`	Integer	Float	Float	18 - 6.5 = 11.5
`SUB`	Float	Integer	Float	18.5 - 7 = 11.5
`MULT`	Integer	Integer	Integer	5 * 11 = 55
`MULT`	Float	Float	Float	3.5 * 12.0 = 42.0

`MULT`	Integer	Float	Float	3 * 12.5 = 37.5
`MULT`	Float	Integer	Float	3.5 * 12 = 42.0
`MULT`	String	Integer	String	"Boo" * 3 = "BooBooBoo"
`MULT`	Integer	String	String	3 * "Boo" = "BooBooBoo"
`DIV`	Integer	Integer	Integer	7 / 2 = 3
`DIV`	Float	Float	Float	7.5 / 3.0 = 2.5
`DIV`	Integer	Float	Float	7 / 2.0 = 3.5
`DIV`	Float	Integer	Float	7.0 / 2 = 3.5

Any combination of types not listed above (e.g., dividing a float by a string) is a runtime error, which means that the program terminates with an error message. Additionally, there are two scenarios that are runtime errors, even though the types are permissible.

- Division by zero (i.e., a `DIV` statement where the operand is either an integer or floating-point zero).
- Negative multiplication of a string (i.e., a `MULT` statement where one operand is a string and the other is a negative integer).

Subroutines

There are no functions or methods in Grin, but there is a simplified mechanism called a *subroutine*. A subroutine is a sequence of Grin statements that can be "called" by executing a `GOSUB` statement. `GOSUB` is much like `GOTO`; it causes execution to jump either by a given number of lines or to a label. However, `GOSUB` also causes Grin to remember where it jumped from. Subsequently, when a `RETURN` statement is reached, execution continues at the line following the `GOSUB` statement that caused the jump. Here's an example:

```
LET A 1
GOSUB 4
PRINT A
PRINT B
END
LET A 2
LET B 3
RETURN
.
```

In the program above, line 1 is executed first, setting the value of `A` to `1`. Next, a `GOSUB` statement is reached. Execution jumps to line 6 (4 greater than the line 2 it appears on), but Grin also remembers that when a `RETURN` statement is reached, execution should jump back to the line following the `GOSUB` — in this case, line 3. Line 6 is executed next, setting `A` to `2`, then line 7 sets `B` to `3`. Now, we reach a `RETURN` statement, causing execution to jump back to the line number that we're remembering — line 3. Line 3 prints the value of `A` (which is `2`), then line 4 prints the value of `B` (which is `3`). Next, we reach line 5, which is an `END` statement, so the program ends.

Subroutines can be used very similarly to Python functions or methods, except they do not take parameters or return a value. Consider the following example, which contains a subroutine that prints the values of `A`, `B`, and `C` each time it's called:

```
LET A 3
GOSUB "PRINTABC"
LET B 4
GOSUB "PRINTABC"
LET C 5
GOSUB "PRINTABC"
```

```
          LET A 1
          GOSUB "PRINTABC"
          END
PRINTABC:  PRINT A
          PRINT B
          PRINT C
          RETURN
          .
```

Subroutines can call other subroutines, meaning that two or more `GOSUB`s may be reached before a `RETURN` is reached. The rules for this are very similar to functions that call other functions in Python; for each `GOSUB` that is reached, Grin will remember the line to which it should return. When a `RETURN` is reached, execution will move to the line remembered from the most recent `GOSUB`. Here's an example.

```
LET A 1
GOSUB 5
PRINT A
END
LET A 3
RETURN
PRINT A
LET A 2
GOSUB -4
PRINT A
RETURN
.
```

In this example, execution begins at line 1 by setting the variable `A` to `1`. Next, we jump to line 7 with a `GOSUB`, but remember that we should jump back to line 3 when we encounter a `RETURN`. Line 7 prints `A` (which is `1`), then line 8 changes `A`'s value to `2`. Now we've reached line 9, which is another `GOSUB` statement. At this point, execution will jump to line 5, but we'll also need to remember to jump back to the line following this `GOSUB` — line 10 — when we reach a `RETURN`. But we also need to remember the line from the previous `GOSUB` — line 3.

Line 5 sets `A` to `3`, then we encounter our first `RETURN` statement. We're remembering two lines — line 3 and line 10. But line 10 is the most recently remembered line, so execution jumps to line 10. Line 10 prints `A` (which is `3`). Now, we encounter another `RETURN` statement on line 11. We're remembering the line 3 from the first `GOSUB`. So, execution jumps to line 3, printing `A` (which is still `3`), then ending the program on line 4.

So, the output of this program is as follows.

```
1
3
3
```

Like `GOTO` statements, `GOSUB` statements are not permitted to jump beyond the boundaries of the program or to non-existent labels, nor can they jump to the same line they came from. If such a `GOSUB` statement is encountered while a program is executed, the program terminates with an error message.

It is also an error for a `RETURN` statement to be encountered when there has been no previous `GOSUB`. The Grin program will immediately terminate and print an error message in this case, as well.

Conditionally altering control flow

Grin provides no precise equivalent of Python's `if` statement, but it does offer a form of conditionality, in the sense that both `GOTO` and `GOSUB` statements can operate conditionally. Optionally, after the target of a `GOTO` or `GOSUB`, we can write the word `IF`, followed by a *comparison expression* that compares two values — literal values or the values in variables — with the result of that comparison determining whether the statement should cause execution to jump.

```
LET A 3
LET B 5
GOTO 2 IF A < 4
PRINT A
PRINT B
.
```

In the program above, the variables `A` and `B` are given the values `3` and `5`, respectively. A `GOTO` statement compares `A` to 4. Since `A` is less than 4, the `GOTO` statement jumps to line 5 — 2 lines beyond itself — and `B` is printed, but `A` is not. The output of the program is as follows.

```
5
```

Both `GOTO` and `GOSUB` can be executed conditionally in this way. The comparison can use one of these six relational operators.

- `<` (less than)
- `<=` (less than or equal to)
- `>` (greater than)
- `>=` (greater than or equal to)
- `=` (equal to)
- `<>` (not equal to)

The types of values being compared partly determine the result of their comparison.

- Integers can be compared to each other, with the results you'd expect.
- Floats can be compared to each other, with the results you'd expect.
- Integers can be compared to floats, with the integer converted temporarily to a float prior to comparison.
- Strings can be compared to each other, with the results being determined *lexicographically* (i.e., by the same rules that Python uses to compare strings).

Comparisons between any other pair of values (e.g., integers to strings) result in runtime errors.

Getting started

Near the top of this project write-up, you'll find a link to a Git repository that provides the starting point for this project. Using the instructions you followed in Project 0 (in the section titled *Starting a new project*), create a new PyCharm project using that Git repository; you'll do your work within that PyCharm project.

Acquainting yourself with the provided code

There's one bit of good news right off the bat: You aren't implementing this project from scratch. Some of the details have been implemented already, and they've been provided to you in their entirety, including the unit tests used to test them. While you won't need to have an expert understanding of every line of that code, you'll need to gain some familiarity with the "public" parts of it (i.e., you'll need to understand the problems solved by the provided code and how to use it), like you would with the Python libraries you've likely used in your prior coursework.

Once you create your new PyCharm project, you'll notice that they're organized similarly to the code in Project 2, with multiple packages used to separate different areas of the program's functionality.

- In the project directory, there is a `project3.py` file, which is the "main" module for your Grin interpreter. It's mainly empty; you'll need to write it.
- A directory named `grin` contains a Python package that can be imported "in bulk".
  - When you write `import grin` in `project3.py` (or any other module outside of the package), the `grin` namespace will contain everything that's imported by `grin`'s `__init__.py` script.
  - Each of the modules in the `grin` package defines a global value named `__all__`, which is a list of the names of what that module "exports" (i.e., what will be imported when `__init__.py` writes `from grin.lexing`

  `import *` is whatever is listed in `lexing.py`'s `__all__` value).
    - The nitty-gritty details of packages in Python are described in Python's documentation, but you won't need to understand more than what's been said here.
- A directory named `tests`, which, in turn, contains a directory named `grin`, in which you'll find unit tests for the provided modules in the `grin` package.

Have a look around what's been provided. You'll find that each file contains documentation describing its purpose, and that each one also describes what ways (if any) you'll need to change them, though most of what you'll be writing will be in new files you'll be adding within the `grin` package and its corresponding test directory.

The program

To satisfy this project's requirements, you'll be building your own Grin interpreter, using the provided code as a starting point. The most basic requirements that your program will need to meet are the following ones.

- The program reads lines of input from the standard input (i.e., from the Python shell), with each line treated as a single Grin statement. Do not print a prompt or anything else; read this input before writing any output. When a line containing only the end-of-program marker is reached, that's considered to be the end of the Grin program, so no more input is read (unless the Grin program itself uses `INNUM` or `INSTR` statements).
- If any errors are encountered while parsing the input, the program prints the corresponding error message — but not a Python traceback — and the program ends.
- After successfully reading an entire Grin program as input, the program interprets the Grin statements, reads their input from the standard input (i.e., from the Python shell), and prints their output to the standard output (i.e., to the Python shell).
- If a runtime error occurs while interpreting the Grin statements (e.g., `GOTO 0`, adding an integer and a string, etc.), the program prints the corresponding error message — but not a Python traceback — and the program ends.
- For a Grin program that executes successfully to completion, your program should generate no output except for what was printed by Grin statements.

Designing your interpreter

As the size of a program increases, one of the most difficult obstacles that inexperienced programmers face is their ability to keep separate issues isolated from one another, so they can work on one problem, get all the way to the bottom of it, and then move on to another. This is sometimes referred to as *separation of concerns*, one of the primary strategies for which is to break a large program into a set of smaller pieces. The obvious mechanism for breaking up a program in Python is the use of classes and functions, though the finesse is in deciding where the seams between those classes and functions should be.

The temptation, especially for novices, is always to try to think about the complete picture, since this strategy works well for the short programs that you write when you're first starting out. As programs become larger, confusion naturally sets in, as the complete picture can be difficult to keep in your brain all at once. Even moderately small Python programs are typically built out of many interacting parts and encompass a great deal of complexity. My complete Grin interpreter has around a dozen modules and several hundred unit tests. (Yours may have fewer, because I implemented a couple of features that I haven't assigned, but this gives you a rough idea about size.) Now, before you freak out, bear in mind that many of those modules contain relatively short classes, a few relatively short functions, and so on. I opted to write more modules with less code in each, so that I could concentrate my efforts on implementing and testing each one largely in the absence of the others.

This project will encourage you to begin thinking about your programs the same way, which will give you the ability to write much larger programs than you could before, as well as enable you to be able to write unit tests at a level of depth you weren't able to do previously.

Design requirements

As you work on your interpreter, you'll want to keep the following requirements in mind.

- As much of the code as possible will need to be written in modules in the `grin` package, with your `project3` module (i.e., the "main" module of your project) being a thin layer that launches the execution of the rest of it.
    - Think of the `project3` module as having one simple job: Knowing that input and output travel through the Python shell, so the rest of the program doesn't need to know that.
- When two issues could be handled in separate modules in the `grin` package, they should be.
    - The provided code, for example, separates the concept of lexing from the concept of parsing; even though they're related, each focuses primarily on one or the other. Even the idea of a "location within the text of the program"

has its own separate type, implemented in its own separate module.

We don't have specific requirements about how many modules you'll need to have, or precisely the way you break your program up, but we'll be evaluating whether you've approached the problem in a way that keeps separate concerns separate.

Within your modules, you'll need to look for opportunities to use classes and inheritance wherever appropriate, both because they're topics we've been exploring in some depth in lecture, and because this problem is one that's amenable to being solved with them, since there are behavioral similarities that can be implemented once and reused using inheritance.

The basic concepts underlying the interpreter

While it's certainly not the case that I start every design by thinking about it in its entirety, one way to start thinking about this particular problem is to brainstorm about the concepts that underlie it. You can then think about the way those concepts fit together, and the ways you might be able to keep them separate. In general, separate concepts should be kept separate until they can't be — which might be quite the opposite of the way you've approached design in your past, because it's not a technique that pays off until programs grow to sizes like this one.

- There are multiple kinds of *statements* in a Grin program. They can all be executed, but different things happen when they're executed. Some things happen the same way for all of them; other things happen the same way for more than one kind of statement.
- There are multiple kinds of *values* in a Grin program: integers, floats, and strings. We can access their values and we can update them, but the rules for updating them are different depending on their types.
- There are multiple kinds of *comparisons* within jump statements (`GOTO` and `GOSUB`), one corresponding to each of the relational operators.
- There will need to be somewhere that keeps track of the *program state*, which consists of a few things.
    - The line number of the statement that's currently executing.
    - The line numbers associated with any labels.
    - The values of any variables.
    - The line numbers being remembered because of `GOSUB` statements.

Each of those concepts could potentially be implemented in a module, and by one or more classes. Don't feel as though you need to build an entire design in your head at once; focus instead on one problem that you can isolate from the others, get your head around that problem, implement (and test) something that you think solves it, and move forward from there. Allow yourself the freedom to be wrong sometimes — not every design idea will pan out, but when you have Git backing you up, you can give up on a bad idea by simply rolling back your changes to the most recent "stable ground" commit.

Unit testing

Along with your Grin interpreter, you will be required to write unit tests, implemented using the `unittest` module in the Python standard library, and covering as much of your interpreter as is practical.

Note that how you design aspects of your interpreter has a positive impact on whether you can unit test it, as well as how hard you might have to work to do it; that's one of the reasons why you're aiming to write multiple modules in the `grin` package, and to tackle separate issues separately. For example, the fact that lexing and parsing are handled separately in the provided code means they can be tested separately; the fact that the result of parsing is a sequence of opaque tokens (instead of strings that need to be parsed again later) makes the code that uses the results of parsing similarly easier to test, since the tests can be written in terms of those higher-level tokens.

There is not a strict requirement around code coverage measurement, nor a specific number of tests that must be written, but we'll be evaluating whether your design accommodates your ability to test it, and whether you've written unit tests that substantially cover the portions that can be tested. (Isolating code that has side effects — such as reading and writing text in the Python shell — can go a long way toward making your program more testable.) The provided tests are there partly to give you an idea of what a reasonably complete set of unit tests look like; your goal is to do likewise.

Grin quick reference

Here is a list of all of the Grin statements (and their different variants) that should be supported by your interpreter, with a brief description of the effect of each.

Statement	*Description*
`LET var value`	Changes the value of the variable *var* to the given *value*, which will either be a literal value or the name of another variable.

`PRINT value`	Prints the given *value* to the console, where *value* will be either a literal value or the name of a variable.
`INNUM var`	Reads a number (either integer or floating-point) into the variable *var*, which must be the name of a variable.
`INSTR var`	Reads a line of text and stores it (as a string) in the variable *var*, which must be the name of a variable.
`ADD var value`	Adds the given *value* to the value of the variable *var*, where *value* will be either a literal value or the name of another variable.
`SUB var value`	Subtracts the given *value* from the value of the variable *var*, where *value* will be either a literal value or the name of another variable.
`MULT var value`	Multiplies the value of the variable *var* by the given *value*, where *value* will be either a literal value or the name of another variable.
`DIV var value`	Divides the given *value* to the value of the variable *var*, where *value* will be either a literal value or the name of another variable.
`GOTO target`	Jumps execution of the program to the given *target*, which will be an integer specifying a relative number of lines or a string containing a label.
`GOTO target IF value1 op value2`	Jumps execution of the program to the given *target*, but only if the values of *value1* and *value2* compare true using the relational operator *op* (`=`, `<>`, `<`, `<=`, `>`, `>=`). If the comparison is false, the statement has no effect.
`GOSUB target`	Temporarily jumps execution of the program to the given *target*, which will be an integer specifying a relative number of lines or a string containing a label. A subsequent `RETURN` statement will cause execution to jump back to the line followed the `GOSUB`.
`GOSUB target IF value1 op value2`	Temporarily jumps execution of the program to the given *target*, but only if the values of *value1* and *value2* compare true using the relational operator *op* (=, <>, <, <=, >, >=). If the comparison is false, the statement has no effect.

`RETURN`	Jumps execution of the program back to the line following the most recently-executed `GOSUB` statement.
`END`	Ends the program immediately.
`.`	Special marker that indicates the end of the program text. Behaves as an `END` statement when encountered.

Sanity-checking your output

We are providing a tool that you can use to sanity check whether you've followed the basic requirements above. It will only give you a "passing" result in these circmustances.

- It's possible to run your program by executing a correctly named module (`project3.py`), spelled and capitalized correctly.
- Executing that module is enough to execute your program.
- Your program reads its input and generates character-for-character correct input for one simple test scenario.

It should be noted that there are many additional tests you'll be want to perform, and that there are many additional tests that we'll be using when we grade your project. The way to understand the sanity checker's output is to think of it this way: Just because the sanity checker says your program passes doesn't mean it's close to perfect, but if you cannot get the sanity checker to report that your program passes, it surely will not pass all of our automated tests (and may well fail all of them).

You'll find the sanity checker in your project directory, in a file named `project3_sanitycheck.py`. Run that program like you would any other, and it will report a result.

Limitations

You can use the Python standard library where appropriate in this project, but you will otherwise not be able to use code written by anyone else other than you. Notably, this includes third-party libraries (i.e., those that are not part of Python's standard library); colloquially, if we have to install something other than Python, Git, and PyCharm in order for your program to work, it's considered off-limits.

Preparing your submission

When you're ready to submit your work, run the provided `prepare_submission.py` script, as you did in prior projects, which will create a Git bundle from the Git repository in your project directory; that Git bundle will be your submission.

Verifying your bundle before submission

If you're feeling unsure of whether your bundle is complete and correct, you can verify it by creating a new PyCharm project from it, as you did in Project 0. (You'll want to create this project in a different directory from your project directory, so it's separate and isolated.) Afterward, you should see the files in their final form, and the **Git** tab in PyCharm should show your entire commit history. If so, you're in business; go ahead and submit your work.

Deliverables

Submit your `project3.bundle` file (and no others) to Canvas. There are a few rules to be aware of.

- When grading your program, we'll grade only the most recent submission. We will not negotiate about which submission will be graded, or, for example, grade multiple of your submissions and "take the highest score."
- When grading your program, we'll grade only the most recent commit on the `main` branch, except to the extent that we'll examine prior commits when evaluating your overall process. We will not negotiate about which commit will be graded, or, for example, grade multiple of your commits and "take the highest score."
- You're responsible for submitting the version of your project that you want graded prior to the deadline. Contacting us afterward and telling us that you accidentally submitted the wrong version will not be grounds for a resubmission under any circumstances.
- You're responsible for making a submission in order to receive credit, which means you'll want to be sure that you've remembered to submit your work and verified in Canvas that it's been received. A later claim of having forgotten to submit your work or having misremembered the due date will not be grounds for a resubmission under any circumstances.
- The determination of whether your work has been submitted before the deadline is the time it was submitted to Canvas. Neither timestamps on local copies of your files nor timestamps on commits in your Git repository or in other

places (e.g., emails or online storage) are considered evidence of completion prior to the deadline under any circumstances.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link. Beyond the late work deadline described there, we will no longer accept submissions.

What do I do if Canvas adjusts my filename?

Canvas will sometimes modify your filenames when you submit them (e.g., by adding a numbering scheme like **-1** or a long sequence of hexadecimal digits to its name). In general, this is fine; as long as the file you submitted has the correct name prior to submission, we'll be able to obtain it with that same name, even if Canvas adjusts it.