Computer Science @ The University of Waterloo

CS 116
Course Specific Style Guide

Last Updated: 2025.04.25

1 Introduction

The code you submit for assignments, as with all code you write, can be made more readable and useful by paying attention to style. This includes the placement of comments, whitespace, indentation, and choice of variable and function names. None of these things affect the execution of a program, but they affect its readability and extensibility. As in writing English prose, the goal is communication, and you need to think of the needs of the reader. This is especially important when the reader is assigning you a grade.

This document will help explain the course specific changes needed for CS 116. This document will also include specific examples outlining how styled code should look for this course.

2 Assignment Formatting Changes

With Python, functions that we write can now do more than simply produce a value. With mutation, a function can change the value of list, dictionary, or object arguments, or have other effects. While allowing programs to do even more, the introduction of mutation requires changes to some steps of our design recipe, along with the introduction of a new step. These changes will be introduced module by module following this preamble.

2.1 Warning About Copying and Pasting

Copying and pasting can make your assignment unmarkable. It may lead to characters appearing in code that our auto graders cannot read correctly and hence will be marked as incorrect. Do not include anything other than ASCII characters (characters found on a standard US keyboard) into your text files.

2.2 Whitespace, Indentation and Layout

As Python is an indentation language, whitespace will largely be forced by the structure of the language. Be consistent and either use two or four spaces for indentation throughout all of your assignments. Do not mix tab characters and spaces as it will cause errors when running in Python. Always use spaces (check your IDE for instructions as to how to make your tab character type in spaces instead of tab characters). Also note that `import` calls should appear one line after the header and should be listed in alphabetical order. Two blank lines should follow these calls. Two blank lines should also be between function definitions.

If the question asks you to write more than one function, the file should contain them in the order specified by the assignment. Follow the convention of placing helper functions above the assignment function(s) they are helping. Remember that the goal is to make it easier for the reader to determine where your functions are located.

When to start a new line (hit enter or return) is a matter of judgment. Try not to let your lines get longer than about 70 characters, and definitely no longer than 80 characters. You do not want your code to look too horizontal or too vertical. You may extend a line by using a backslash character \ in Python or if a previous line ends with a open parenthesis that has no corresponding closing parenthesis, you may continue the line by hitting enter (and eventually typing the closing parenthesis, finishing the extended line). In most cases, the 80 character line limit will not be as heavily enforced when writing tests.

Style marks may be deducted if you exceed 80 characters on any line of your assignment.

2.3 Comments

In Python, use `##` or `#` to indicate the start of a comment. Use `##` for full-line comments and `#` for in-line comments. The rest of the line will be ignored. Use in-line comments sparingly. If you are using standard design recipes and templates, and following the rest of the guidelines here, you should not need many additional comments. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

Racket programs were generally short enough that the design recipe steps provided sufficient documentation about the program. It is more common to add comments in the function body when using imperative languages like Python. While there will not usually be marks assigned for internal comments, it can be helpful to add comments so that the marker can understand your intentions more clearly. In particular, comments can be useful to explain the role of local variables, if statements, and loops. Comments should address the *why* and not *how* a program works.

2.4 Headers

As with Racket, your files should start with a header to identify yourself, the term, the assignment and the problem. We recommend using the following:

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Assignment 03, Problem 4
## **************************************************
##
```

2.5 Function Header - Python's Docstring

The function header will now be the first thing in a function. Documentation follows the header.

```
def circle_area(r)
  '''
  Documentation Here
  '''
```

Python programmers can attach documentation to functions (which the Python `help` function displays) using documentation strings, or docstrings. We will use a docstring for our purpose statements, contract and requirements, effects statements and examples. It is placed directly after the function header. Everything will be included in a single string. As this string will extend over multiple lines, we will use three single quotes to signify the beginning and end of our docstring for a function.

2.6 Constant (Variable) and Function Identifiers

2.6.1 Naming Functions, Parameters, Variables and Constants

Function and parameter names should be meaningful, but not awkwardly long nor cryptically short. The first priority should be to choose a meaningful name. Names like `salary` or `remittance` would be appropriate in a program that calculates taxes. Sometimes a function will consume values that don't have a meaning attached, for example, a function that calculates the maximum of two numbers. In that case, choose names that reflect the structure of the data. That is, `n` is for numbers, `i` for integers, `lst` for a list or `lon` for a list of numbers, and so on. Names that are proper nouns like Newton should always be capitalized (though this will not be enforced). Otherwise, use the following function naming conventions:

- The dash ("-") character cannot be used as an identifier in Python. You should instead use an underscore ("_") where a dash would have been used (*e.g.,* `tax_rate`).
- In Python you may use one of the following:
  - Snake Case: use only lowercase letters and separate words with an underscore (*e.g.,* `top_bracket_amount`).
  - Camel Case: begin with a lower/upper case letter; words that follow are capitalized (*e.g.,* `topBracketAmount` or if capitalizing the first word, `TopBracketAmount`).

It is important to pick one convention and to be consistent with your naming convention in any given file. In some cases, this will mean following the convention given by the assignment (for example, if the function you are to write is `my_fun`, then your helper functions should also use snake case). Assignments will almost always follow snake case conventions, so pay close attention!

2.6.2 Constants

Constants should be used to improve your code in the following ways:

- To improve the readability of your code by avoiding "magic" numbers. For example, if you have code dealing with tax rates like Ontario's HST, you might want a constant such as `taxes` or `hst` and have this value set to `0.13`.
- To improve flexibility and allow easier updating of special values. If the value of `taxes` changes, it is much easier to make one change to the definition of the constant than to search through an entire program for the value

`0.13`. When this value is found, it may not be obvious if it refers to the tax rate, or whether it serves a different purpose and should not be changed.

- To define values for testing and examples. As values used in testing and examples become more complicated (*e.g.,* lists, classes, dictionaries), it can be very helpful to define named constants to be used in multiple tests and examples.

2.7 Summary

- Use two comment symbols for full-line comments and use one comment symbol for in-line comments, and use them sparingly inside the body of functions.
- Provide a file header for your assignments.
- Place appropriate amounts of blank lines between various code elements.
- Make it clear where function blocks begin and end.
- Order your functions appropriately.
- Avoid overly horizontal or vertical code layout, and use reasonable line lengths.
- Choose meaningful identifier names and follow our naming conventions, ensuring that the conventions are consistent.
- Avoid use of "magic numbers."

Style marks may be deducted if you have poor headers, identifier names, whitespace, indentation or layout.

3 The Design Recipe: Functions

Warning! This style guide will be used for all assessments (*i.e.*, assignments and exams).

We hope you will use the design recipe as part of the process of working out your solutions to assignment questions. If you hand in only code, even if it works perfectly, you will earn only a fraction of the marks available. Elements of the design recipe help us to understand your code.

3.1 Design Recipe Sample

In your final version, the content including and after `# <---` should be removed and is included here to highlight the different components of the design recipe.

```
import check


def sum_of_squares(p1, p2):                       # <--- Function Header
  '''
  Returns the sum of the squares of p1 and p2.    # <--- Purpose

  sum_of_squares: Int Int -> Nat                  # <--- Contract

  Examples:                                       # <--- Examples
    sum_of_squares(0, 0) => 0
    sum_of_squares(3, 4) => 25
  '''

  return p1 * p1 + p2 * p2                        # <--- Function Body

## Examples as Tests:                             # <--- Tests
check.expect("Example 1, boundary case", sum_of_squares(0, 0), 0)
check.expect("Example 2, generic case", sum_of_squares(3, 4), 25)

## Other Tests:
check.expect("Test one 0", sum_of_squares(0, 2), 4)
check.expect("Test negative, generic case", sum_of_squares(-2, 7), 53)
```

3.2 Purpose

```
'''
Returns the sum of the squares of p1 and p2.       # <--- Purpose
'''
```

Your purpose statements should briefly explain what the function does, using parameter names to show the relationship between the input and the function's actions. The actions may include returning a value, mutations, the use of `input` and `print`, as well as any file operations. There are two small changes when writing our purpose statements:

- Use "return" rather than "produce", to reflect the use of the `return` statement in Python functions, when a function returns a value.
- As the purpose statement follows immediately after the header, we will now omit the function call from our purpose statement.

As before,

- The purpose starts with an example of how the function is applied, which uses the same parameter names used in the function header.
- The purpose should be written in complete sentences; capitalize the first word of each sentence.
- Do not write the word "purpose".
- The description must include the names of the parameters in the purpose to make it clear what they mean and how they relate to what the function does (choosing meaningful parameter names helps also). You do not need to include parameter types and requirements in your purpose statement — the contract already contains that information.
- If the description requires more than one line, "indent" the next line of the purpose by two or three spaces.
- If you find the purpose of one of your helper functions is too long or too complicated, you might want to reconsider your approach by using a different helper function or perhaps using more than one helper function.

3.3 Contract

```
'''
sum-of-squares: Int Int -> Nat
'''
```

The contract contains the name of the function, the types of the parameters it consumes, and the type of the value it returns. The contract is analogous to functions defined mathematically that map from a domain to a codomain (or more loosely to the range of the function).

There are a few significant omissions from the Racket types. Note that:

- Python does not have a Symbol type. You should use strings or numbers in the role of symbolic constants as needed. Python does have an enumerated type, but it is not used in CS 116.
- Python does not have a Character type. Instead, use strings of length one.
- Python does not use the `Num` type. If a value can be either an integer or non-integer value, use `(anyof Int Float)`.
- Note that we will maintain a `Nat` data type because it is convenient but Python does not have a native natural number type (natural numbers are integers according to Python).

If there are additional restrictions on the consumed types, continue to use a requirements statement following the contract. If there are any requirements on data read in from a file or from standard input, these should be included in the requirements statement as well.

3.3.1 Additional Contract Requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section after the contract. For example, this can be used to indicate an `Int` must be in a specific range, a `Str` must be of a specific length, or that a `(listof ...)` cannot be empty. Single requirements can be on one line. Multiple requirements should start on a separate line and each line should be indented by two or three spaces.

```
'''
quot: Nat Nat -> Nat
Requires:
  n1 >= 0
  n2 > 0
'''
```

3.3.2 Python Types

The following table lists the valid Python types.

`Any`	Any value is acceptable.
`(anyof T1 T2...)`	Mixed data types. For example, `(anyof Int Str)` can be either an `Int` or a `Str`; `(anyof Int False)` can be either an `Int` or the value `False`, but not `True`.
`Float`	Any non-integer/real value.
`Int`	Integers: `...-2, -1, 0, 1, 2...`
`Nat`	Natural numbers (non-negative integers): `0, 1, 2...`
`None`	The `None` value designates when a function does not include a `return` statement or when it consumes no parameters.
`Str`	String (*e.g.,*`"Hello There"`, `"a string"`)
`X, Y, ...`	Matching types to indicate parameters must be of the same type. For example, in the following contract, the `X` can be any type, but all of the `X`'s must be the same type: `my-fn:  X (listof X) -> X`
`Bool` (Module 2)	Boolean values (`True` and `False`)
`(listof T)` (Module 4)	A list of **arbitrary length** with elements of type `T`, where `T` can be any valid type. For example, `(listof Any)`, `(listof Int)`, `(listof (anyof Int Str))`.
`(list T1 T2...)` (Module 4)	A list of **fixed length** with elements of type `T1`, `T2`, etc. For example, `(list Int Str)` always has two elements: an `Int` (first) and a `Str` (second).
`User_Defined` (Module 9)	For user-defined class types. Capitalize your user-defined types.
`(dictof T1 T2)` (Module 9)	A dictionary with keys of type `T1` and associated values of type `T2`.

3.4 Examples

```
'''
Examples:
  sum_of_squares(0, 0) => 0
  sum_of_squares(3, 4) => 25
'''
```

The examples should be chosen to illustrate "typical" uses of the function and to illuminate some of the difficulties to be faced in writing it. Examples should cover each case described in the data definition for the type consumed by the function. The examples do not have to cover all the cases that the code addresses; that is the job of the tests, which are designed after the code is written. It is very useful to write your examples before you start writing your code. These examples will help you to organize your thoughts about what *exactly* you expect your function to do. You might be surprised by how much of a difference this makes. Unless otherwise stated, you may use examples given in assignments for your documentation but you are encouraged to also write your own.

Unlike in Racket, examples in Python cannot be written as test code using the provided `check` module. Unfortunately, function calls to the functions in the `check` module cannot come before the actual function definitions. Therefore, instead of writing examples as code, we will include them as part of our function's docstring. The notation to write an example is to use `fcn_call(parameter_data) => output`. The format of the example depends on whether or not the function has any effects. Examples that fit on one line can be written as so. Otherwise, the first sentence should start on a new line and be indented by two or three spaces. Remember to be consistent with your indentation within the docstring!

Examples should cover edge cases whenever possible (for example, empty lists, smallest and/or largest possible cases and so on). For recursive data, your examples **must** include *each* base case and at least one recursive case.

3.5 Testing

```
## Examples as Tests:
check.expect("Example 1, boundary case", sum_of_squares(0, 0), 0)
check.expect("Example 2, generic case", sum_of_squares(3, 4), 25)

## Other Tests:
check.expect("Test one 0", sum_of_squares(0, 2), 4)
check.expect("Test negative, generic case", sum_of_squares(-2, 7), 53)
```

It is extremely important that you submit your code and check your basic tests to make sure the format of your submission is readable by the autotests!

Python does not present us with a function like `check-expect` in Racket for testing our programs. To emulate this functionality, you can download the `check.py` module from the course website. This module contains several functions designed to make the process of testing Python code straightforward, so that you can focus on choosing the best set of test cases. You must save the module in the same folder as your program, and include the line `import check` at the beginning of each Python file that uses the module. You do not need to submit `check.py` when you submit your assignments.

Our tests for most functions will consist of several parts; you only need to include the parts that are relevant to the function you are testing. These additions will be introduced in each module as necessary. What follows is sufficient to test code in the beginning of the course.

1. If there are any global state variables to use in the test, set them to specific values. Separate these variables from the constants defined at the top of the file.
2. Write a brief description of the test as the descriptive label in the testing function.
3. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`), followed by more `check` calls, one for each mutated value to be checked (Module 4).

Some additional information about testing follows.

- You should always set the values of every global state variable in every test before calling any `check` functions, in case your function inadvertently mutates their values. In other words, define all global state variables before making any calls to `check` functions.

- The two main functions included in `check` are `check.expect` and `check.within`; these functions will handle the actual testing of your code. You should only use `check.within` if you expect your code to produce a floating point number or an object containing a floating point number. In every other case, you should use `check.expect`. When testing a function that produces nothing, you should use `check.expect` with `None` as the expected value.
  - `check.expect` consumes three values: a string (a label for the test, such as “Question 1, Test 6”, or a description of the test case), a value to test, and an expected value. You will pass the test if the value to test equals the expected value; otherwise, it will print a message that includes both the value to test and the expected value, so that you can see the difference.
  - `check.within` consumes four values: a string (a label/description for the test), a value to test, an expected value, and a tolerance. You will pass the test if the value to test and the expected value are close to each other (to be specific, if the absolute value of their difference is less than or equal to the tolerance); otherwise, it will print a message that includes both the value to test and the expected value, so that you can compare the results. The standard tolerance in the course is 0.00001.
- Include labels in comments to denote different “sections" of your testing suite. It is often helpful to separate the example cases from the rest of the test cases.

Examples **must** be rephrased as tests in Python since the `check` module does not read the docstring for tests (see the exemplars in later sections).

3.5.1 Testing Tips

Make sure that your tests are actually testing every part of the code. For example, if a conditional expression has three possible outcomes, ensure that you have tests that check each of the possible outcomes. Furthermore, your tests should be directed; each one should aim at a particular case, or section of code. Some people write tests that use a large amount of data; this is not necessarily the best idea, because if they fail, it is difficult to figure out why. Others write lots of tests, but have several tests in a row that do essentially the same thing. It's not a question of quantity, but of quality. You should design a small, comprehensive test suite.

Never figure out the answers to your tests by running your own code. Work out the correct answers independently (*e.g.,* by hand).

Parameter Type	**Consider trying these values:**
`Float`	Positive, negative, zero, non-integer values, specific boundaries, small and/or large values
`Int`	Positive, negative, zero
`Bool`	`True`, `False`
`Str`	Empty string (`""`), length 1, length $>$ 1, extra whitespace, different character types, etc.
`(anyof ...)`	Values for each possible type
`(listof T)`, `(dictof T)`	Empty (`[]`, `{}`), length 1, length $>$ 1, duplicate values, special situations, etc.
`Class` object	Special values for each field (structures), and for each possibility (mixed types)

Recall there are two types of tests: black box tests and white box tests. Black box tests are done with no knowledge of what the implementation of a function is; these are done based solely on what the description of the function does. White box tests are done after doing the implementation with part of the goal to make sure every line of code is tested by some test. Both types of tests are vital to creating a good testing suite for your program. When in doubt while creating tests, imagine you were trying to break your code and try to write tests that would check those edge cases.

3.6 Helper Functions

- Do not use the word "helper" in your function name: use a descriptive function name.
- Only a purpose and a contract (and effects, if relevant) is necessary for design recipe for helper functions.
- You are not required to provide tests for your helper functions, but often it is a very good idea to ensure that they work properly.

- In the past, we have seen students avoid writing helper functions because they did not want to provide documentation for them. This is a bad habit that we strongly discourage! Writing good helper functions is an essential skill in software development and having to write a purpose and contract should not discourage you from writing a helper function.
- Marks may be deducted if you are not using helper functions when appropriate.
- Helper functions should be placed before the required function(s) in your submission.
- Helper functions may *not* be defined locally in your Python programs.

3.7 Summary

- Design recipe elements are the same as in Racket, but with some changes.
- Use **returns** instead of **produces** in purpose statements.
- Python does not have `Sym` or `Num` data types. Use `Float` in place of `Num` where appropriate.
- Examples **must** be repeated as tests since our `check` module does not test examples.
- For recursive data, your examples **must** include *each* base case and at least one recursive case. Examples should cover edge cases whenever possible (for example, empty lists).
- Include `import check` at the top of your code in order to use our testing module.
- Do not use the word **helper** in a helper function name.
- Helper functions do not need examples and tests but must have a purpose, contract and body.

3.8 Final Thoughts

In what follows, we go module by module and discuss specific language features that will be added as needed. The next several pages also include lots of examples for you to get some experience with writing code in Python. Be sure to refer back to this guide periodically throughout the course.

4 Module 1

4.1 Module 1: Basic Example with Check Expect

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 1 Sample
## **************************************************
##

import check
import math


## Constants
lucky_seven = 7


def add_7(n):
  '''
  Returns the integer n + 7.

  add_7: Int -> Int

  Examples:
    add_7(0) => 7
    add_7(-7) => 0
    add_7(100) => 107
  '''

  return n + lucky_seven

## Examples as Tests:
check.expect("Testing first example", add_7(0), 7)
check.expect("Testing second example", add_7(-7), 0)
check.expect("Testing third example", add_7(100), 107)

## Other Tests:
check.expect("Testing large negative", add_7(-1000), -993)
check.expect("Testing small positive", add_7(1), 8)
check.expect("Testing huge value", add_7(10 ** 10), 10 ** 10 + lucky_seven)
```

4.2 Module 1: Basic Example with Check Within

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 1 Sample
## **************************************************
##

import check
import math


## Here is where constants would go if necessary.


def circle_area(r):
  '''
  Returns the area of a circle of given radius r.

  circle_area: Float -> Float
  Requires: r >= 0.0

  Examples:
    circle_area(0.0) => 0.0
    circle_area(1.0) => 3.141592653589
  '''

  return math.pi * r * r

tol = 0.00001 # Place tolerance constant with tests.
## Examples as Tests:
check.within("Example 1: Zero area", circle_area(0.0), 0.0, tol)
check.within("Example 2: Pi", circle_area(1.0), 3.1415926, tol)

## Other Tests:
check.within("Small value", circle_area(0.01), 0.0003141592, tol)
check.within("Large value", circle_area(100), 31415.926535, tol)
```

4.3 Module 1: Basic Example with Strings

```
##
## *************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 1 Sample
## *************************************************
##

import check


def cat(s, t)
  '''
  Returns the concatenation of two strings s and t.

  cat: Str Str -> Str

  Examples:
    cat("", "") => ""
    cat("top", "hat") => "tophat"
  '''

  return s + t

## Examples as Tests:
check.expect("Testing first example", cat("", ""), "")
check.expect("Testing second example", cat("top", "hat"), "tophat")

## Other Tests:
check.expect("Test one empty", cat("pot", ""), "pot")
check.expect("Test other empty", cat("", "cat"), "cat")
check.expect("Test duplicate", cat("too", "too"), "tootoo")
```

5 Module 2

5.1 Module 2: Recursive Example

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 2 Sample
## **************************************************
##

import check


def factorial(n):
  '''
  Returns the product of all the integers from 1 to n.

  factorial: Nat -> Nat

  Examples:
    factorial(0) => 1
    factorial(1) => 1
    factorial(5) => 120
  '''

  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)

## Examples as Tests:
check.expect("Example 1: Zero factorial", factorial(0), 1)
check.expect("Example 2: One factorial", factorial(1), 1)
check.expect("Example 3: Five factorial", factorial(5), 120)

## Other Tests:
check.expect("Two factorial", factorial(2), 2)
check.expect("Seven factorial", factorial(7), 5040)
twenty_factorial = 20*19*18*17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2
check.expect("twenty factorial", factorial(20), twenty_factorial)
```

6 Module 3

In module 3, we introduce side effects for the first time. This will bring some changes to the design recipe.

6.1 Purpose

Actions functions perform may now include returning a value and the use of `input` and `print`.

6.2 Effects

When a function's role involves anything in addition to or instead of returning a value, it must be noted in an effects statement. This includes:

- Printing to screen
- Taking input from user

Effects should be written **after** the purpose statement but **before** the contract. A single blank line should precede and follow the effects section.

6.3 Examples

We also need to make additions to examples to account for printing and input from keyboard.

- If the function involves printing and/or input from the keyboard, then these side effects need to be explained in the examples. Notice that the return value of the function is also provided.

  ```
  '''
  Examples:
    If the user enters Waterloo and Ontario when prompted by
    enter_hometown() => None
    then the following is printed to the screen:
    Waterloo, Ontario
  '''
  ```

  The descriptions here need not be exact if the print output is convoluted but should provide a reasonable summary of what will be displayed.

- If the function returns a value *and* has side effects, you will need to use a combination of descriptions.

```
'''
Examples:
  If the user enters Smith when prompted,
  enter_new_last("Li", "Ha") => "Li Smith"
  and the following is printed to the screen:
  Ha, Li
  Smith, Li
'''
```

6.4 Testing

We need to make some additions to testing as well to account for side effects. Some special `check` functions will be useful! These are clarified below.

1. If there are any global state variables to use in the test, set them to specific values. Separate these variables from the constants defined at the top of the file.
2. Write a brief description of the test as the descriptive label in the testing function.
3. If you expect user input from the keyboard, include a call to `check.set_input` *before* your testing function.
4. If you expect your function to print anything to the screen, include a call to `check.set_screen` or `check.set_print_exact` *before* your testing function and *after* any potential calls to `check.set_input`.
5. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`), followed by more `check` calls, one for each mutated value to be checked (Module 4).

6.4.1 Testing Keyboard Input

If your function uses keyboard input, you will need to use the command `check.set_input` before running the test. This function consumes strings corresponding to the input that will be used instead of waiting for data to be typed in when the function is called. This means that you do not need to do any manual input when you run your tests; `check.set_input` will take care of it for you. You will get an error if you do not have enough strings in your call to `check.set_input` or if there are too many; you must have exactly the correct number of strings.

6.4.2 Testing Print Output

If the value to test is a call to a function that prints to the screen, you have two choices for checking the printing: `check.set_screen` and `check.set_print_exact`.

You can use `check.set_print_exact` (which consumes one string for each line you expect your function to print to the screen) before running the test. When the test is run, in addition to comparing the actual and expected returned values, the strings passed to `check.set_print_exact` are compared to what is actually printed by your function call. Two messages will be printed: one for the returned value and one regarding the printed strings. This command will *not* test for output printed to the screen from an `input` call (*i.e.,* input prompts).

Alternatively, you can use `check.set_screen` (which consumes a string describing what you expect your function to print) before running the test. Print output will have **no effect** on whether the test is passed or failed. When you call `check.set_screen`, the next test you run will print both the output of the function you are testing, and the expected output you gave to `check.set_screen`. You need to visually compare the output to make sure it is correct. As you are the one doing the visual comparison, you are also the one to determine the format of the string passed to `check.set_screen`.

It does not matter which function you use to test print output; the choice is yours! We recommend using `check.set_print_exact` when the expected print output is smaller and only takes up a few lines, and we recommend using `check.set_screen` when the expected print output is larger and may be tedious to type out line by line (*e.g.,* ASCII drawings, game simulations). That being said, try to be consistent with which function you are using for a given problem.

6.5 Module 3: Printing Example

Notice below that you can choose to include that the function returns `None` or omit this since the contract captures this information.

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 3 Sample
## ***************************************************
```

```
##

import check


def print_it_three_times(s):
  '''
  Prints s on three lines, once per line.

  Effects: Prints to the screen

  print_it_three_times: Str -> None

  Examples:
    print_it_three_times("") => None
    and three blank lines are printed.

    print_it_three_times("a") => None
    and a is printed once on each of three lines.
  '''

  print(s)
  print(s)
  print(s)

## Examples as Tests:
check.set_print_exact("", "", "")
check.expect("Example 1: Empty string", print_it_three_times(""), None)

check.set_print_exact("a", "a", "a")
check.expect("Example 2: Single character", print_it_three_times("a"), None)

## Examples as Tests (set_screen):
check.set_screen("Three blank lines")
check.expect("Example 1: Empty string", print_it_three_times(""), None)
check.set_screen("a on three separate lines")
check.expect("Example 2: Single character", print_it_three_times("a"), None)

## Other Tests:
check.set_print_exact("CS 116", "CS 116", "CS 116")
check.expect("Test random string", print_it_three_times("CS 116"), None)

## Other Tests (set_screen):
check.set_screen("CS 116 on three separate lines")
check.expect("Test random string", print_it_three_times("CS 116"), None)
```

6.6 Module 3: Keyboard Input Example

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 3 Sample
## ***************************************************
##

import check


def diff():
  '''
  Reads two integers from the keyboard and prints the difference.

  Effects:
    Reads input from keyboard
    Prints to the screen

  diff: None -> None

  Examples:
    If the user enters enters 5 and 3 after
    diff() => None
    is called, then 2 is printed to the screen.

    If the user enters enters 0 and 0 after
    diff() => None
    is called, then 0 is printed to the screen.
  '''

  x = int(input("Enter an integer: "))
  y = int(input("Enter another integer: "))
  print(x - y)

## Examples as Tests:
check.set_input("5", "3")
check.set_print_exact("2")
check.expect("Testing first example", diff(), None)

check.set_input("0", "0")
check.set_print_exact("0")
check.expect("Testing second example", diff(), None)

## Other Tests:
check.set_input("1", "3")
check.set_print_exact("-2")
check.expect("Negative answer", diff(), None)

check.set_input("-1", "-3")
check.set_print_exact("2")
check.expect("Negative inputs", diff(), None)
```

6.7 Module 3: Keyboard Input and Screen Output

```
##
## ****************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 3 Sample
## ****************************************************
##

import check


## Constants
multiply = "double"
divide = "half"
multiply_str = "string"


def mixed_fn(n, action):
  '''
  Returns 2 * n if action is "double", n / 2 if action is "half",
    and returns None otherwise. Further, if action is "string",
    the user is prompted to enter a string and then n copies of
    that string are printed on one line. For any other action,
    "Invalid action" is printed to the screen.

  Effects:
    If action == "string":
      Reads input from keyboard
      Prints to the screen
    If action != "string", "half" or "double":
      Prints to screen

  mixed_fn: Nat Str -> (anyof Int Float None)

  Examples:
    mixed_fn(2, "double") => 4
    mixed_fn(11, "half") => 5.5

    mixed_fn(6, "oops") => None
      and Invalid action is printed to the screen.

    If the user enters a after calling
      mixed_fn(5, "string") => None
      then aaaaa is printed to the screen.
  '''

  if action == multiply:
    return 2 * n
  elif action == divide:
    return n / 2
  elif action == multiply_str:
    s = input("Enter a non-empty string: ")
    print(s * n)
  else:
    print("Invalid action")

tol = 0.00001
## Examples as Tests:
check.expect("Try double", mixed_fn(2, "double"), 4)

check.within("Try half with odd", mixed_fn(11, "half"), 5.5, tol)

check.set_print_exact("Invalid action")
check.expect("Invalid action", mixed_fn(6, "oops"), None)

check.set_input("a")
check.set_print_exact("aaaaa")
check.expect("Try string", mixed_fn(5, "string"), None)

## Other Tests:
check.within("Try half with even", mixed_fn(20, "half"), 10.0, tol)

check.set_input("hello")
check.set_print_exact("hellohellohello")
check.expect("Try string", mixed_fn(3, "string"), None)

check.set_print_exact("Invalid action")
check.expect("Invalid action", mixed_fn(2, "DOUBLE"), None)

## A test using set_screen:
check.set_input("word")
check.set_screen("word repeated 10 times without spaces")
check.expect("Try string", mixed_fn(10, "string"), None)
```

7 Module 4

In module 4, we introduce lists, mutable objects and mutation for the first time. We need to update some design recipe sections accordingly.

7.1 Purpose

Actions functions perform may include returning a value, mutating a passed parameter, and the use of `input` and `print`.

7.2 Effects

Our effects now can include:

- Printing to screen
- Taking input from user
- Mutating a passed parameter (*e.g.,* a list, a dictionary (Module 9), a class object (Module 9))

7.3 Examples

We also need to make additions to examples to account for mutations. If the function involves mutation, the example needs to reflect what is true before and after the function is called.

```
'''
Examples:
  If lst1 = [1, -2, 3, 4], then
  mult_by(lst1, 5) => None
  and lst1 = [5, -10, 15, 20].
'''
```

7.4 Testing

We now include mutation in our testing routine.

1. If there are any global state variables to use in the test, set them to specific values. Separate these variables from the constants defined at the top of the file.

2. Write a brief description of the test as the descriptive label in the testing function.
3. If you expect user input from the keyboard, include a call to `check.set_input` *before* your testing function.
4. If you expect your function to print anything to the screen, include a call to `check.set_screen` or `check.set_print_exact` *before* your testing function and *after* any potential calls to `check.set_input`.
5. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`), followed by **more `check` calls, one for each mutated value to be checked.**

In the case of lists with `check.within`, a test will fail if any one of the components of the list is not within the specified tolerance.

You should only mutate parameters in problems that explicitly state to do so. You should also test for a lack of mutation for your questions (though you will not be explicitly graded on this) as our backend tests might test for a lack of mutation of given parameters.

7.5 Module 4: Mutation Example

Notice below how the helper function does not need examples.

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 4 Sample
## ***************************************************
##

import check


def add_ones_to_evens_by_position(lst, pos):
  '''
  Mutates lst by adding 1 to each even value starting from position pos.

  Effects: Mutates lst

  add_ones_to_evens_by_position: (listof Int) Nat -> None
  '''

  if pos < len(lst):
    if lst[pos] % 2 == 0:
      lst[pos] = lst[pos] + 1
    add_ones_to_evens_by_position(lst, pos+1)


def add_one_to_evens(lst):
  '''
  Mutates lst by adding 1 to each even value.

  Effects: Mutates lst

  add_one_to_evens: (listof Int) -> None

  Examples:
    If L = [], then
    add_one_to_evens(L) => None
    and L = [].

    If L = [3, 5, -18, 1, 0], then
    add_one_to_evens(L) => None
    and L = [3, 5, -17, 1, 1].
  '''

  add_ones_to_evens_by_position(lst, 0)

## Examples as Tests:
L = []
check.expect("Empty list", add_one_to_evens(L), None) # Check return
check.expect("Empty list, mutation", L, []) # Check mutation

L = [3, 5, -18, 1, 0]
check.expect("Generic example", add_one_to_evens(L), None) # Check return
check.expect("Generic example, mutation", L, [3, 5, -17, 1, 1]) # Check mutation

## Other Tests:
L = [2]
check.expect("One even number", add_one_to_evens(L), None) # Check return
check.expect("One even number, mutation", L, [3]) # Check mutation

L = [7]
check.expect("One odd number", add_one_to_evens(L), None) # Check return
check.expect("One odd number, mutation", L, [7]) # Check mutation

L = [1, 4, 5, 2, 4, 6, 7, 12]
check.expect("General case", add_one_to_evens(L), None) # Check return
check.expect("General case, mutation", L, [1, 5, 5, 3, 5, 7, 7, 13]) # Check mutation
```

7.6 Module 4: Non-Mutation Example

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 4 Sample
## **************************************************
##

import check


def add_evens(lst):
  '''
  Returns the sum of all even numbers in lst.

  add_evens: (listof Int) -> Nat

  Examples:
    add_evens([]) => 0
    add_evens([3, 5, -18, 1, 0]) => -18
  '''

  add = 0
  if lst == []:
    return 0
  elif lst[0] % 2 == 0:
    add = lst[0]
  return add + add_evens(lst[1:])

## Examples as Tests:
L = []
check.expect("Empty list", add_one_to_evens(L), None) # Check return
check.expect("Empty list, mutation", L, []) # Check non-mutation

L = [3, 5, -18, 1, 0]
check.expect("Generic example", add_one_to_evens(L), None) # Check return
check.expect("Generic example, mutation", L, [3, 5, -18, 1, 0]) # Check non-mutation

## Other Tests:
L = [2]
check.expect("One even number", add_one_to_evens(L), None) # Check return
check.expect("One even number, mutation", L, [2]) # Check non-mutation

L = [7]
check.expect("One odd number", add_one_to_evens(L), None) # Check return
check.expect("One odd number, mutation", L, [7]) # Check non-mutation

L = [1, 4, 5, 2, 4, 6, 7, 12]
check.expect("General case", add_one_to_evens(L), None) # Check return
check.expect("General case, mutation", L, [1, 4, 5, 2, 4, 6, 7, 12]) # Check non-mutation
```

8 Module 5

8.1 Module 5: Accumulative Recursion Example

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 5 Sample
## **************************************************
##

import check


def remember_fact(product, n0):
  '''
  Returns the product of all the integers from 1 to n0
    multiplied by the value in product.

  remember_fact: Nat Nat -> Nat
  '''

  if n0 <= 1:
    return product
  else:
    return remember_fact(product * n0, n0 - 1)


def factorial(n):
  '''
  Returns the product of all the integers from 1 to n.

  factorial: Nat -> Nat

  Examples:
    factorial(0) => 1
    factorial(1) => 1
    factorial(5) => 120
  '''

  return remember_fact(1, n)

## Examples as Tests:
check.expect("Example 1: Zero factorial", factorial(0), 1)
check.expect("Example 2: One factorial", factorial(1), 1)
check.expect("Example 3: Five factorial", factorial(5), 120)

## Other Tests:
check.expect("Two factorial", factorial(2), 2)
check.expect("Seven factorial", factorial(7), 5040)
```

```
twenty_factorial = 20*19*18*17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2
check.expect("Twenty factorial", factorial(20), twenty_factorial)
```

8.2 Module 5: Accumulative Recursion with Lists

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 5 Sample
## **************************************************
##

import check


def remember_evens(n, acc):
  '''
  Returns even numbers from n to 0 appended
    to the intermediary answer in acc.

  Effects: Mutates acc

  remember_fact: Nat (listof Nat) -> (listof Nat)
  '''

  if n < 0:
    return acc
  elif n % 2 == 0:
    acc.append(n)
  return remember_evens(n - 1, acc)


def ret_evens(n):
  '''
  Returns a list of even numbers from n to 0.

  ret_evens: Nat -> (listof Nat)

  Examples:
    ret_evens(0) => [0]
    ret_evens(5) => [4, 2, 0]
  '''

  return remember_evens(n, [])

## Examples as Tests:
check.expect("Example 1: Zero", ret_evens(0), [0])
check.expect("Example 2: Five", ret_evens(5), [4, 2, 0])

## Other Tests:
check.expect("Test random", ret_evens(6), [6, 4, 2, 0])
check.expect("Test one", ret_evens(1), [0])
```

8.3 Module 5: Generative Recursion Example

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 5 Sample
## ***************************************************
##

import check


def is_palindrome(s):
  '''
  Returns True if and only if s is a palindrome and False otherwise.

  is_palindrome: Str -> Bool

  Examples:
    is_palindrome("") => True
    is_palindrome("a") => True
    is_palindrome("abba") => True
    is_palindrome("abca") => False
  '''

  if len(s) < 2:
    return True
  else:
    return s[0] == s[-1] and is_palindrome(s[1:-1])

## Examples as Tests:
check.expect("Example 1: Empty", is_palindrome(""), True)
check.expect("Example 2: Single Character", is_palindrome("a"), True)
check.expect("Example 3: Palindrome", is_palindrome("abba"), True)
check.expect("Example 4: Non-palindrome", is_palindrome("abca"), False)

## Other Tests:
check.expect("Long Palindrome", is_palindrome("aba" * 20), True)
check.expect("Short Palindrome", is_palindrome("aa"), True)
check.expect("Almost Palindrome", is_palindrome("aaaaabcaaaaaa"), False)
```

8.3.1 Module 5: Local Helper Function

As a rule in this course we generally discourage the use of local helper functions. The following example helps to show why. Notice how the helper function awkwardly doesn't include the parameter `n` but must reference it to make sense of what the function does. The local helper function can access parameters that are part of the containing function.

```
##
## ***************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 5 Local Helper Function
## ***************************************************
##

import check


def fib(n):
    '''
    Returns nth Fibonacci number

    fib: Nat -> Nat

    Examples:
        fib(0) => 0
        fib(1) => 1
        fib(10) => 55
    '''

    def acc(n0, last, prev):
        '''
        Returns nth Fibonacci number, where last is the n0th,
        and prev is (n0-1)th

        acc: Nat Nat Nat -> Nat
        '''
        if n0 >= n:
            return last
        else:
            return acc(n0 + 1, last + prev, last)

    # Body of fib4
    if n == 0:
        return 0
    else:
        return acc(1,1,0)


##Examples:
```

```
check.expect("Example 1: f0", fib(0), 0)
check.expect("Example 2: f1", fib(1), 1)
check.expect("Example 3: f10", fib(10), 55)

##Tests:

check.expect("f2", fib(2), 1)
check.expect("f3", fib(3), 2)
check.expect("f5", fib(5), 5)
```

9 Module 9

In module 9, we introduce the dictionary data type as well as classes. Testing for dictionaries is analogous to lists; in particular, you may need to check if a dictionary has been mutated. Use `(dictof key value)` in contracts.

9.1 Python Classes

We can define new types using a Python class. Each class should contain "magic" methods (`__init__`, `__repr__`, `__eq__`) to make the class easy to use. Class names should begin with a capital letter. Following the class definition, the fields should be listed in a docstring consisting of the field name and the type of the field in parentheses. The word `Fields:` should be in line with the docstring quotes, followed by 2 or 3 spaces (consistently) and the field names with types afterwards in parentheses. Requirements on the parameters should then follow as done below:

```
'''
Fields:
  hour (Nat)
  minute (Nat)
  second (Nat)

Requires:
  0 <= hour < 24
  0 <= minute, second < 60
'''
```

The description for `__init__` should include a purpose statement and effects:

```
def __init__(self, h, m, s):
  '''
  Constructor: Create a Time object by calling Time(h, m, s).
  Effects: Mutates self
  '''
```

followed by the contract and requirements:

```
  '''
  __init__: Time Nat Nat Nat -> None
  Requires: 0 <= h < 24, and 0 <= m, s < 60
  '''
```

The usual conventions for our design recipe still apply for other class methods, including `__repr__` and `__eq__`. Examples for magic methods are **not required** but are required for all other class methods you write. When writing functions that consume or return objects of a user-defined class, the standard style guidelines still apply.

When writing class methods (functions inside the class definition that can be called using Python's dot notation), remember that the first parameter is always called `self` and its type in the contract is the (capitalized) name of the class. The purpose, effects, contracts and requirements, and examples should be written following the standard style guidelines. You can (and should) use `self` in the purpose statement. Note, though, that your tests for class methods **cannot be included in the class**. They must be defined after the full class definition.

Note that if a class definition is given to you but is missing documentation, then you do not need to fill in design recipe elements unless specifically told to do so.

9.2 Module 9: Dictionaries Example

Recall that the order of key-value pairs in a dictionary does not matter.

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 9 Sample
## ***************************************************
##

import check


def character_count(sentence):
  '''
  Returns a dictionary of the character count of letters in sentence.

  character_count: Str -> (dictof Str Nat)

  Examples:
    character_count('') => {}
    character_count('a') => {'a': 1}
    character_count('banana') => {'a': 3, 'b': 1, 'n': 2}
  '''

  characters = {}
  for char in sentence:
    if char in characters:
      characters[char] = characters[char] + 1
    else:
      characters[char] = 1
  return characters

## Examples as Tests:
check.expect("Example 1: Empty", character_count(''), {})
check.expect("Example 2: Singleton", character_count('a'), {'a': 1})
check.expect("Example 3: Typical", character_count('banana'),
  {'n': 2, 'a': 3, 'b': 1})
check.expect("Example 3: Typical", character_count('banana'),
  {'a': 3, 'b': 1, 'n': 2})

## Other Tests:
check.expect("Spaces", character_count('hi mom'),
  {'h': 1, 'i': 1, ' ': 1, 'm': 2, 'o': 1})
check.expect("Large", character_count('a' * 10000), {'a': 10000})
```

9.3 Module 9: Classes Example

```
##
## **************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 9 Sample
## **************************************************
##

import check


## Constants
seconds_per_minute = 60
minutes_per_hour = 60
seconds_per_hour = seconds_per_minute * minutes_per_hour


class Time:
  '''
  Fields:
    hour (Nat)
    minute (Nat)
    second (Nat)

  Requires:
    0 <= hour < 24
    0 <= minute, second < 60
  '''

  def __init__(self, h, m, s):
    '''
    Constructor: Create a Time object by calling Time(h, m, s).

    Effects: Mutates self

    __init__: Time Nat Nat Nat -> None
    Requires: 0 <= h < 24, and 0 <= m, s < 60
    '''

    self.hour = h
    self.minute = m
    self.second = s
  def __repr__(self):
    '''
    Returns a string representation of self. Implicitly called
      by print(t), where t is of type Time.

    __repr__: Time -> Str
    '''

    ## The 0:0=2d means display 2 digits.
    return "{0:0=2d}:{1:0=2d}:{2:0=2d}".format(
      self.hour, self.minute, self.hour)


  def __eq__(self, other):
    '''
    Returns True if self and other are considered equal, and
      False otherwise. Implicitly called for t1 == t2 or t1 != t2,
      where t1 is a Time value, and t2 is any type.

    __eq__: Time Any -> Bool
    '''

    return isinstance(other, Time) and \
      self.hour == other.hour and \
      self.minute == other.minute and \
      self.second == other.second


  ## Note this is a class method (defined within the class).
  def time_to_seconds(self):
    '''
    Returns the number of seconds since midnight for self.

    time_to_seconds: Time -> Nat

    Examples:
      If midnight = Time(0, 0, 0),
      then midnight.time_to_seconds() => 0.

      If just_before_midnight = Time(23, 59, 59),
      then just_before_midnight.time_to_seconds() => 86399.
    '''

    return (seconds_per_hour * self.hour) + \
      seconds_per_minute * self.minute + self.second


## Note this is a function (defined outside of the class).
def earlier(time1, time2):
  '''
  Returns True if and only if time1 occurs before time2, and False otherwise.

  earlier: Time Time -> Bool
  Examples:
    just_before_midnight = Time(23, 59, 59)
    noon = Time(12, 0, 0)

    earlier(noon, just_before_midnight) => True
    earlier(just_before_midnight, noon) => False
  '''

  return time1.time_to_seconds() < time2.time_to_seconds()

## A sample test for __init__
midnight = Time(0, 0, 0)

check.expect("Example __init__ hour", midnight.hour, 0)
check.expect("Example __init__ minute", midnight.minute, 0)
check.expect("Example __init__ second", midnight.second, 0)

## A sample test for __repr__
midnight = Time(0, 0, 0)

check.set_print_exact("00:00:00")
check.expect("Example __repr__", print(midnight), None)

check.expect("Example __repr__ Alternate",
  str(midnight), "00:00:00"))

## A sample test for __eq__
midnight = Time(0, 0, 0)
midnight2 = Time(0, 0, 0)
notmidnight = Time(0, 0, 1)

check.expect("Example __eq__", midnight == midnight2, True)
check.expect("Example __eq__", midnight == notmidnight, False)

## Examples for time_to_seconds:
midnight = Time(0, 0, 0)
just_before_midnight = Time (23, 59, 59)

check.expect("Midnight: min answer", midnight.time_to_seconds(), 0)
check.expect("Just before midnight: max answer",
  just_before_midnight.time_to_seconds(), 86399)

## Tests for time_to_seconds:
noon = Time(12, 0, 0)
eight_thirty = Time(8, 30, 0)
eight_thirty_and_one = Time(8, 30, 1)

check.expect("Noon", noon.time_to_seconds(), 12 * seconds_per_hour)
check.expect("Eight Thirty", eight_thirty.time_to_seconds(),
  17 * seconds_per_hour // 2)
check.expect("Eight Thirty and 1", eight_thirty_and_one.time_to_seconds(),
  17 * seconds_per_hour // 2 + 1)
```

```
## Examples for earlier:
midnight = Time(0, 0, 0)
just_before_midnight = Time (23, 59, 59)

check.expect("Before", earlier(noon, just_before_midnight), True)
check.expect("After", earlier(just_before_midnight, noon), False)

## Tests for earlier:
midnight = Time(0, 0, 0)
just_before_midnight = Time (23, 59, 59)
noon = Time(12, 0, 0)
eight_thirty = Time(8, 30, 0)
eight_thirty_and_one = Time(8, 30, 1)

check.expect("Before", earlier(midnight, eight_thirty), True)
check.expect("After", earlier(eight_thirty, midnight), False)
check.expect("Just before", earlier(eight_thirty, eight_thirty_and_one),
  True)
check.expect("Just after", earlier(eight_thirty_and_one, eight_thirty),
  False)
check.expect("Same time", earlier(eight_thirty_and_one, eight_thirty_and_one),
  False)
```

10 Module 10

In module 10, we introduce file input and output. This once again comes with updates to design recipe sections.

10.1 Purpose

Actions functions perform may include returning a value, mutating a passed parameter, the use of `input` and `print`, as well as any file operations.

10.2 Contracts

If a specific format for a file is required, this needs to be stated in the requirements. Functions that are passed strings to files must also be ensured to exist in the current directory.

10.3 Effects

Our full list of possible effects is as follows:

- Printing to screen
- Taking input from user
- Mutating a passed parameter
- Reading a file
- Writing to a file

10.4 Examples

Do your best to describe what is written to files and what is read from files.

```
'''
Examples:
  If the user enters Smith when prompted,
  enter_new_last("Li", "Ha") => "Li Smith"
  and the following is written to "NameChanges.txt":
  Ha, Li
  Smith, Li
'''
```

10.5 Testing

At last, we can now state our full testing procedure!

1. If there are any global state variables to use in the test, set them to specific values. Separate these variables from the constants defined at the top of the file.
2. Write a brief description of the test as the descriptive label in the testing function.
3. If you expect user input from the keyboard, include a call to `check.set_input` *before* your testing function.
4. If you expect your function to print anything to the screen, include a call to `check.set_screen` or `check.set_print_exact` *before* your testing function and *after* any potential calls to `check.set_input`.
5. If you expect your function to write to any files, include a call to `check.set_file_exact` *before* your testing function and *after* any potential calls to `check.set_input`, `check.set_print_exact` or `check.set_screen`.
6. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`), followed by more `check` calls, one for each mutated value to be checked.

10.5.1 Testing File Input

If your function reads from a file, you will need to create the file (using a text editor like Wing IDE or using the Add File button in EdX) and save it in the same directory as your `aXXqY.py` files. You do not need to submit these files when you submit your code, but any test that reads from a file should include a comment with a description of what is contained in the files read in that test.

10.5.2 Testing File Output

If your function writes to a file, you will need to use the command `check.set_file_exact` before running the test. The function consumes two strings: the first is the name of the file that will be created by the function call in step 6, and the second is the name of a file identical to the one you expect to be created by the test. You will need to create the second file yourself using a text editor.

The next call to `check.expect` or `check.within` will compare the two files in addition to comparing the expected and actual returned values. If the files are exactly the same, the test will print nothing; if they differ in any way, the test will print which lines don't match, and will print the first pair of differing lines for you to compare.

There is also a function `check.set_file` that has the same parameters as `check.set_file_exact`, which sets up a comparison of the two files when all whitespace is removed from both files. Use this function only if explicitly told to on an assignment.

10.6 Module 10: File Input and Output Example

```
##
## ***************************************************
## Rita Sanchez (12345678)
## CS 116 Winter 2025
## Module 10 Sample
## ***************************************************
##

import check


def file_filter(fname, minimum):
  '''
  Opens the file fname, reads in each integer, and writes each integer > minimum
    to a new file, "summary.txt".

  Effects:
    Reads the file called fname
    Writes to file called "summary.txt"

  file_filter: Str Int -> None
  Requires:
    0 <= minimum <= 100
    fname exists in the current directory

  Examples:
    If "empty.txt" is empty, then
    file_filter("empty.txt", 1) => None
    will create an empty file named summary.txt.

    If "ex2.txt" contains 35, 75, 50, 90 (one per line), then
    file_filter("ex2.txt", 50) =>
    will create a file named "summary.txt" containing
    75, 90 (one per line).
  '''
  infile = open(fname, "r")
  lst = infile.readlines()
  infile.close()
  outfile = open("summary.txt", "w")
  for line in lst:
    if int(line.strip()) > minimum:
      outfile.write(line)
  outfile.close()

## Test 1: empty file (example 1)
check.set_file_exact("summary.txt", "empty.txt")
check.expect("T1", file_filter("empty.txt", 40), None)

## Test 2: small file (example 2)
## eg2-summary contains 75 and 90 once per line.
check.set_file_exact("summary.txt", "eg2-summary.txt")
check.expect("T2", file_filter("ex2.txt", 50), None)

## Test 3: file contains one value, it is greater than minimum
check.set_file_exact("summary.txt", "one-value.txt")
check.expect("T3", file_filter("one-value.txt", 20), None)

## Test 4: file contains one value, it is less than minimum
check.set_file_exact("summary.txt", "empty.txt")
check.expect("T4", file_filter("one-value.txt", 80), None)

## Test 5: file contains one value, it is equal to minimum
check.set_file_exact("summary.txt", "empty.txt")
check.expect("T5", file_filter("one-value.txt", 50), None)

## Test 6: file contains 1 - 30 on separate lines
check.set_file_exact("summary.txt", "sixteen-thirty.txt")
check.expect("T6", file_filter("thirty.txt", 15), None)
```