# Style Guide for CS 231 and CS 234

Naomi Nishimura

February 8, 2022

## 1 Introduction

This document specifies the style required for Python programs written for the courses CS 231 and CS 234. It assumes familiarity with the design recipe, as discussed in detail in first-year CS courses, and uses standards set out for Python in PEP 8 – Style Guide for Python Code.

You may find discrepancies among the styles used in textbooks, in course materials (where examples may be condensed for the sake of concision), and in this guide. Unless specified otherwise, you should always use the style presented in this guide on all work submitted for the course.

This guide includes general principles for the formatting of assignments (Section 2), details of the design recipe (Section 3), guidelines for definitions of user-designed data types, including class definitions (Section 4), and sample files that satisfy style requirements (Section 5).

For both courses, programming questions are intended to help reinforce course concepts. Consequently, it is your demonstration of course concepts that is most important. Code that passes automatic tests but does not use the correct paradigm (for CS 231) or the specified implementation (for CS 234) will not receive full marks.

Here are the main points, which may differ from requirements for earlier courses:

- Documentation for the design recipe may make use of either docstrings or comments. Lecture content may omit documentation entirely, in order to save space.
- You are expected to provide or complete docstrings for all class definitions. A provided template may provide some, but not necessarily all, of the information required for a complete and correct docstring.
- Helper functions are always permissible, including additional private methods for classes. However, make sure that your helper functions are not being used to subvert the intention of the question, which will be to have you practice using course concepts.

- Departure from best programming practices may be acceptable in situations where the priority is demonstrating understanding of course concepts. For example, in CS 234, repetition of code may be the best choice to separate the roles of user and provider in implementations of various operations of an ADT. Similarly, in CS 231, the specified paradigm might result in an inefficient algorithm.

- Examples and tests are highly recommended, but are not required. Tests should be placed in a testing file that is not submitted, as mentioned in Section 3.7.4.

# 2 Assignment format

Each assignment should start with an assignment header, include documentation as either a docstring or comments, and make use of various techniques for enhancing readability.

A docstring is simply a string that follows a class definition header, function header, or method header. To print the docstring of the class `MyClass`, you can use either `print(MyClass.__doc__)` to see the docstring for the class definition, or `help(MyClass)` to see the docstring as well as all the methods. For a method or function, you can also use `__doc__` or `help` to see the docstring.

Make sure to use the name of the function or method that is used in a function call by the user. For example, for the method `repr`, use `print(repr.__doc__)` or `help(repr)` (not `print(__repr__.__doc__)` or `help(__repr__)`). In contrast, for the method `row_num` of the class `Grid`, use `print(Grid.row_num.__doc__)` or `help(Grid.row_num)`.

## 2.1 Assignment header and order of components

Start each assignment file with an assignment header to identify yourself, the term, the assignment, and the problem. While there is no specific required format, the information should be presented in a clear manner, such as in the example below:

```
##
## ======================================================
##   Ima Student (12345678)
##   CS 231 Spring 2025
##   Assignment 3, P2
## ======================================================
##
```

For an assignment question that asks you to provide a function, the file should contain the following information, in the following order:

- assignment header
- import statements (if any)
- definitions of user-defined types (if any), including documentation
- constant definitions (if any), including documentation
- most helper functions (if any), including documentation
- main program/function, including documentation
- helper functions, such as recursive helper functions paired with a main function that is a nonrecursive shell

For an assignment question that asks you to provide a class definition, such as from a provided code interface file, the file should contain the following information, in the following order:

- assignment header
- import statements (if any)
- constant definitions (if any), including documentation
- class definition, including
  - docstring to document the attributes in the class
  - helper methods (if any), including documentation
  - methods provided in the code interface file, including documentation

Please see Section 3 for details about how to provide documentation for each function or method.

User-defined types (especially classes) are particularly important in CS 234; be sure to read Section 4 carefully to ensure that you follow the proper format.

## 2.2 Documentation

Most of the documentation you provide will appear as part of a docstring, appearing right after a class definition, function header or method header, or in a block of comments at the beginning of a function or method, as dictated by the design recipe. Typically, part of the marks for an assignment will be allocated for such documentation.

If you are provided with a code interface with partial documentation for methods, make sure that the documentation for all methods is complete and that you provide (or complete) a docstring for any class definition.

Additional comments should be used sparingly to indicate your intentions, such as the use of local variables, branching, and loops. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

In Python, use ## or # to indicate the start of a comment. Either is acceptable. The rest of the line will be ignored.

## 2.3 Blank lines, layout, indentation, and line length

Blank lines can be used to group related information and set it apart from other components, such as the documentation for a class definition, function, or method. An alternative way of separating information is the use of a row of =’s or other symbols as a comment, similar to what is used in the assignment header.

If a question asks you to write more than one function, the file should contain them in the order specified by the assignment. Helper functions should be put near the function they are helping. Remember that the goal is to make it easy for the reader to determine where your functions are located.

The use of indentation in Python code reflects not only style but syntax, as levels of indentation are used to indicate the relationships among lines of code. In order to have a visually-pleasing layout, use four spaces for each level of indentation of code. Indentation should also be used in comments in order to indicate that lines belong to the same part of the documentation.

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 can use \ at the end of a line to indicate that the material on the following line is a continuation.

## 2.4 Identifiers

To be consistent with Python conventions, names of functions, methods, and variables should use lower-case letters, with words joined by underscores (e.g. `tax_rate`), names of constants should use upper-case letters, with words joined by underscores (e.g. `DAYS_IN_WEEK`), and names of classes should be capitalized (e.g. `Time`). For attributes or methods that are intended to be private, often the name is preceded by an underscore (e.g. `_data`). (Note: The use of the underscore is for the sake of humans; do not expect the computer to enforce the rules for you.)

Try to choose names for functions and parameters that are descriptive, not so short as to be cryptic, but not so long as to be awkward. The detail required in a name will depend on context: if a function consumes a single number, calling that number `n` is probably fine. Avoid using the name "helper" in the name of a helper function, instead opting for a more descriptive choice.

For readability and ease of debugging, choose variable names that make it clear what type of data is being stored in that variable, such as using `num` in the name of a variable storing a number and `lst` in the name of a variable storing a list. In CS 234, using this convention can help distinguish among different user-defined objects (typically, ADTs).

Recall that constants should be used to improve your code in the following ways:

- Constants improve the readability of your code by avoiding "magic" numbers. Make sure to document the meaning of each constant.
- Constants improve flexibility and allow easier updating of special values.
- Constants can define values for testing and examples. As values used in testing and examples become more complicated (*e.g.,* lists, objects, lists of objects), it can be very helpful to define named constants to be used in multiple tests and examples.

Style marks may be deducted if your code is hard to read due to poor use of headers, identifiers, blank lines, indentation, layout, and line length.

# 3 The design recipe

The design recipe specifies a series of steps to take in creating code; these steps serve both to help you create clear and correct code and to make your intentions clear to the markers. Some of the steps result in code and some in comments. As marks will be assigned to specific steps, do not expect to receive full marks for handing in an assignment that contains only code (even if perfect) and no comments.

You are not required to include examples and tests for your work in either CS 231 or CS 234. However, it is strongly recommended that you use these steps to ensure that you can trust your code to be correct.

The recommended order in which to create the components of a function is not the same as the order in which the components will appear in the final program.

**Recommended order of creation** (also the order of the subsections below):

1. function header
2. contract (including **Requires** section if needed)
3. purpose
4. effects
5. examples
6. function body
7. tests

**Recommended order in the final program, using docstring**:

1. function header
2. purpose (in docstring)
3. effects (in docstring)
4. contract (including **Requires** section if needed) (in docstring)
5. examples (in docstring)

6. function body

**Recommended order in the final program, using a block of comments**:

1. purpose
2. effects
3. contract (including **Requires** section if needed)
4. examples
5. function definition (function header and function body)

Tests should be put in a separate document, as discussed in Section 3.7.4.

## 3.1 Function header [code]

The reason for writing the function header first is so that you have chosen the names of the function and parameters and the order of the parameters. You will be using these in upcoming steps.

## 3.2 Contract [documentation]

The contract is used to clearly specify the types of the inputs and output (if any) of the function. It contains the name of the function, a colon, the types of the arguments it consumes (if any), an arrow (consisting of a hyphen and a greater-than sign), and the type of the value it produces (if any). The value of a call to a function with no return statement is `None`.

```
## function_name: Type1 Type2 ... Typek -> Type
```

The following table lists the abbreviations for Python data types to be used in contracts:

| | |
|---|---|
| `Float` | A non-integer numerical value |
| `Int` | An integer |
| `Bool` | A Boolean value (`True` or `False`) |
| `Str` | A string (*e.g.,* `"Hello"`, `"This is string!!?"`) |
| `None` | The value of a call to a function with no return statement. |

For more complex data types, the following abbreviations are to be used:

| | |
|---|---|
| `(anyof T1 T2 ...Tk)` | A value that can be any of T1 through Tk, where each is a data type or specific value and $k \geq 2$. For example, `(anyof Int Str)` can be either an `Int` or a `Str`. If `False` is used to indicate an unsuccessful result, use `(anyof Int False)` instead of `(anyof Int Bool)` for greater precision. |
| `Any` | A value that can be any data type. |
| `(listof T)` | A list of **arbitrary length** with elements of type T, where T can be any data type. Examples include `(listof Any)`, `(listof Int)`, and `(listof (anyof Int Str))`. |
| `(list T1 T2 ...Tk)` | A list of length $k$ where the first element is of type T1, the second of type T2, and so on. For example, `(list Int Str)` always has two elements: an `Int` (first) and a `Str` (second). |
| `ClassName` | An object of type ClassName, where ClassName is the name of a class. |
| `(dictof T1 T2)` | A dictionary with keys of type T1 and associated values of type T2. |

In addition, letters such as `X` and `Y` can be used to specify that types used in parameters in a contract must be the same. 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`.

If there are important constraints on the parameters that are not fully described in the contract, add a **Requires** section after the contract, where indentation is used for the second and subsequent requirements, if any. 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.

The examples below can be used to indicate a `Float` must be in a specific range, a `Str` must be of a specific length, or that a `(listof ...)` cannot be empty. The first uses a docstring (which appears after the function header or method header) and the second uses a block of comments (which appears before the function header or method header); either format is acceptable.

```
"""
mystery_function(first second third a_string num_list) does something.
mystery_function: Float Float Float Str (listof Float) -> Bool
Requires: 0 < first < second
          third must be non-zero
```

```
            a_string must be of length 3
            num_list must be non-empty and contain distinct elements
                sorted in ascending order
    """

## mystery_function(first second third a_string num_list) does something.
## mystery_function: Float Float Float Str (listof Float) -> Bool
## Requires: 0 < first < second
##           third must be non-zero
##           a_string must be of length 3
##           num_list must be non-empty and contain distinct elements
##               sorted in ascending order
```

## 3.3 Purpose [documentation]

The purpose statement should briefly explain the actions resulting from a function call (such as producing a value, changing a function argument or state variable, or using `input`, `print`, or file operations) using parameter names to show the relationship between the input and the actions. The purpose statement does not need to provide all the details of how each action is performed, but should describe the result. It also does not need to include parameter types or requirements, as these already appear in the contract.

## 3.4 Effects [documentation]

An effects statement is required for any function action other than producing a value. Following the label `Effects:`, explicitly list each state variable or parameter that is changed when the function is called, each use of `input` or `print`, and each file operation. The description of the change itself should be included, as noted above, in the purpose statement for the function.

## 3.5 Examples [documentation]

Examples are optional in CS 231 and CS 234, but highly recommended.

Choose examples that illustrate various possible cases encountered in using the function; for example, for recursive data your examples should include at least one base and one non-base case. Examples are written as comments or as part of the docstring, where the format of the example depends on whether or not the function has any effects.

- If the function produces a value, but has no effects, then the example can be written as a function call followed by a double arrow (typed as an equals

sign followed by a greater than sign) followed by the value of the function call.

```
## Example:
##     combine_str_num("hey", 3) => "hey3"
```

- If the function involves mutation, the example should indicate conditions that are true before and after the function is called.

```
## Example:
##     If lst1 is initially [1, -2, 3, 4]
##     then after the call mult_by(lst1, 5)
##     lst1 = [5, -10, 15, 20]
```

- If the function involves some other effects (reading from keyboard or a file, or writing to screen or a file), then this needs to be explained, in words, in the example as well.

```
## Example:
##     If the user enters Waterloo and Ontario when prompted by
##     enter_hometown(), the following is written to the screen:
##         Waterloo, Ontario
```

- If the function produces a value and has effects, all the information needs to be conveyed.

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

## 3.6 Function body [code]

Only after writing the contract (possibly including requirements), purpose, effects, and examples should you write the function body.

## 3.7 Tests [code]

In CS 231 and CS 234, you will not be submitting tests with your code. Instead, you will creating an extra (optional, but highly recommended) file for your own

use. Do not submit testing files.

For each function, your test suite should be small and comprehensive, containing a single test for each particular case being considered. Taken together, the tests should exercise every part of the code, such as every possible outcome of a conditional expression. Tests are required to check the results of a function, whether values produced, changes to state variables or parameters, or other actions.

Never figure out the answers to your tests by running your own code. Work out the correct answers by hand.

### 3.7.1 The module `check.py`

The module `check.py`, developed for CS 116, provides several functions that can be used in testing Python code. Download `check.py` from the course site and save the module in the same folder as your program.

You will most often test your code using `check.expect`. If you expect your code to produce a floating point number (or if one part of the produced value is a floating point value), use `check.within` instead of `check.expect`. The functions `check.expect` and `check.within` will print `PASSED` if the value produced matches the expected value and `FAILED` otherwise.

- `check.expect` consumes three values: a string (a label for the test, such as "Question 1, empty string"), a value to test, and an expected value. The function will print `PASSED` if the value to test equals the expected value; otherwise, a message is printed that includes both the expected value and the value seen, allowing you to compare the results.

- `check.within` consumes four values: a string (a label of the test, such as "Question 2, even number"), a value to test, an expected value, and a tolerance. The function will print `PASSED` if the value to test and the expected value are close to each other (more specifically, if the absolute value of their difference is less than the tolerance); otherwise, a message is printed that includes both the expected value and the value seen. In the case of lists or objects containing floating point values, the test will fail if any one of these components is not within the specified tolerance.

### 3.7.2 The module `equiv.py`

For output in the form of Python lists (or lists of lists), where the content of the lists, but not their order, is to be tested, you have been provided with a module `equiv.py` with functions for use in such types of tests. Please see the relevant reference page on the course site for more details.

### 3.7.3 Steps in a test

Since a function may have actions other than producing a value, additional steps may be necessary to check the mutation of values. In particular, even if `PASSED` is printed, some of the actions may not be correct. Each test consists of up to four steps, where only the relevant steps are required in any particular test; at a minimum, each test will contains steps 1 and 3.

**Step 1: Write a brief description of the test as a comment.**
Briefly describe the case that is being tested.
If your function reads from a file, the comment should also include a description of the contents of the file. In addition, you will need to create the file (using a text editor like Wing IDE) and save it in the same directory as your assignment solution files. You do not need to submit such files when you submit your code.

**Step 2: Set values of state variables.**
Set each state variable to a specific value, whether or not it appears in the effects. This will allow you to test that the function does not inadvertently mutate values of state variables that are not supposed to change, and that the function does mutate values of state variables that are supposed to change.

**Step 3: Check the value produced by the function.**
Use `check.expect` or `check.within`, whichever is appropriate. When testing a function that does not produce a value, use `check.expect` with `None` as the expected value.

**Step 4: Check the values of state variables and/or parameters (if any).**
In this step you will check every state variable to ensure either that it has been mutated correctly, if it appears in the effects of the function, or that it has not changed, if it does not appear in the effects of the function. You will also check the mutated value of every parameter that is included in the effects of the function. For each such value, use either `check.expect` or `check.within`, whichever is appropriate.

The following chart gives suggestions for tests that might be appropriate for different types of data.

| Parameter type | Consider trying these values |
|---|---|
| `Float` | positive, negative, 0, specific boundaries |
| `Int` | positive, negative, 0, specific boundaries |
| `Bool` | `True`, `False` |
| `Str` | empty string (`""`), length 1, length $> 1$, extra whitespace, different character types |
| `(anyof ...)` | values for each possible type |
| `(listof T)` | empty, length 1, length $> 1$, duplicate values in list, special situations |
| `User_Defined` | special values for each attribute (classes), and for each possibility (mixed types) |

### 3.7.4 Testing files

To test a file, create a testing file that imports both `check.py` and the assignment file to be tested. Your testing file should include all of your tests. Do not submit your testing file, do not submit `check.py`, and do not import `check.py` into any of the files that you submit for your assignments. An example testing file is shown in Section 5.4.

# 4 User-defined data

Each course has provided user-defined data types for your use in the course. For each, you will use the name in contracts, such as the data type `Grid` in CS 231 and the data type `Contiguous` in CS 234. Where appropriate, make sure to use **Requires** sections as well.

## 4.1 Class definitions

CS 234 makes extensive use of code interfaces in the form of class definitions. A sample can be found in Section 5.3.

Associated with each class is a docstring, a string which can be printed to give information about the class. At a minimum, you should include the names of the

attributes and their types; if there are further restrictions on the attributes, add a **Requires** section, as in the sample given in Section 5.2.

Note: For consistency with CS 116, the docstrings for class definitions use the term *field* instead of *attribute*. The two terms can be used interchangeably.

Each method should be documented like a function.

## 4.2 Descriptive definitions

A descriptive definition can be used to define a new user-defined type to improve the readability of contracts and other type definitions.

```
## A StudentID is an Int
## Requires: The value is between 1000000 and 99999999

## A Direction is an (anyof "up" "down" "left" "right")
```

A descriptive definition can also be a mixed type, with more than one possible type. In the following example, we assume that the types `StudentID`, `StaffID`, and `FacultyID` have all been defined.

```
## A CampusID is one of:
## * a StudentID
## * a StaffID
## * a FacultyID
## * "guest"
```

# 5 Samples

In the samples below, notice the order within the documentation for each function, the overall order, the use of blank lines, and the different styles used for assignment headers. Notice also how indentation is used in comments and docstrings such as, for example, a multi-line purpose statement.

## 5.1 Using mutation

Since this example concerns lists, there is an example for both the base case (an empty list) and a non-base case (a non-empty list).

Due to the use of mutation, each test consists of the setting of a state variable, checking that the value of function call is `None`, and then making sure that the state variable has been changed as expected.

```
##
## ************************************************
## *    Ima Student (12345678)                    *
## *    CS 231 Spring 2025                        *
## *    Assignment 03, Problem 1                  *
## ************************************************
##

import check

def add_one_to_evens(int_list):
    """
    add_one_to_evens(int_list) mutates int_list by adding 1
        to each even value.
    Effects: Mutates int_list by adding 1 to each even value.
    add_one_to_evens: (listof Int) -> None
    Examples:
        if int_list = [],  add_1_to_evens(int_list) => None,
            then int_list = [].

        if int_list = [3,5,-18,1,0], add_one_to_evens(int_list)
            => None, then int_list = [3,5,-17,1,1].

    """
    for i in range(len(int_list)):
        if int_list[i]%2==0:
            int_list[i] = int_list[i] + 1

# Test 1: Empty list
list_one = []
check.expect("Q1T1", add_one_to_evens(list_one), None)
check.expect("Q1T1 (list_one)", list_one, [])

# Test 2: List of one even number
list_two = [2]
check.expect("Q1T2", add_one_to_evens(list_two), None)
check.expect("Q1T2 (list_two)", list_two, [3])

# Test 3: List of one odd number
list_three = [7]
check.expect("Q1T3", add_one_to_evens(list_three), None)
check.expect("Q1T3 (list_three)", list_three, [7])

# Test 4: General case
list_four = [1,4,5,2,4,6,7,12]
check.expect("Q1T4", add_one_to_evens(list_four), None)
check.expect("Q1T4 (list_four)", list_four, [1,5,5,3,5,7,7,13])
```

## 5.2 Using a class definition

In this example, notice how the docstring for the class definition includes the names and data types of attributes, as well as additional requirements on the data.

The example includes three methods, which are part of the class definition, as well as two functions. Examples and tests are provided for the functions only.

```
##
## @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
##   Ima Student (12345678)
##   CS 231 Spring 2025
##   Assignment 03, Problem 4
## @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
##

import check

## Constants used to convert values
SECONDS_PER_MINUTE = 60
MINUTES_PER_HOUR = 60
SECONDS_PER_HOUR = SECONDS_PER_MINUTE * MINUTES_PER_HOUR

## Constants used for examples and testing
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)

class Time:
    """
    Fields: hour (Int), minute (Int), second (Int)
    Requires: 0 <= hour < 24
              0 <= minute, second < 60

    """

    def __init__(self, hour, minute, second):
        """
        Time(hour, minute, second) produces a Time object with
            hour hours, minute minutes, and second seconds.
        Effects: Creates a new Time.
        __init__: Int Int Int -> Time
        Requires: 0 <= hour < 24 and 0 <= minute, second < 60

        """
        self.hour = hour
        self.minute = minute
        self.second = second


    def __str__(self):
```

```
        """
        print(self) produces a string with hour, minute, and seconds.
        __str__: Time -> Str

        """
        return "{0:.2}:{1:.2}:{2:.2}".format(self.hour, \
               self.minute, self.second)

    def __eq__(self, other):
        """
        self == other produces True if self and other are equal
            and False otherwise.
        __eq__: Time Any -> Bool

        """
        return instance(other, time) and \
               self.hour == other.hour and \
               self.minute == other.minute and \
               self.second == other.second

def time_to_seconds(t):
    """
    time_to_seconds(t) produces the number of seconds since midnight
        for the Time t.
    time_to_seconds: Time -> Int
    Examples:
        time_to_seconds(MIDNIGHT) => 0
        time_to_seconds(JUST_BEFORE_MIDNIGHT) => 86399

    """
    return (SECONDS_PER_HOUR * t.hour) + \
           SECONDS_PER_MINUTE * t.minute + t.second

## Test 1 for time_to_seconds - NOON
check.expect("Q4-tts-1", time_to_seconds(NOON), 43200)
## Test 2 for time_to_seconds - EIGHT_THIRTY
check.expect("Q4-tts-2", time_to_seconds(EIGHT_THIRTY), 30600)
## Test 3 for time_to_seconds - EIGHT_THIRTY_AND_ONE
check.expect("Q4-tts-3", time_to_seconds(EIGHT_THIRTY_AND_ONE), 30601)

def earlier(time1, time2):
    """
    earlier (time1, time2) determines if time1 occurs before time2.
    earlier: Time Time -> Bool
    Examples:
        earlier(NOON, JUST_BEFORE_MIDNIGHT) => True
        earlier(JUST_BEFORE_MIDNIGHT, NOON) => False

    """
    return time_to_seconds (time1) < time_to_seconds (time2)

## Test 1 for earlier - first earlier than second
check.expect("Q4-e-1", earlier(MIDNIGHT, EIGHT_THIRTY), True)
```

```
## Test 2 for earlier - second earlier than first
check.expect("Q4-e-2", earlier(EIGHT_THIRTY, MIDNIGHT), False)
## Test 3 for earlier - first earlier than second, differ in seconds
check.expect("Q4-e-3", earlier(EIGHT_THIRTY, EIGHT_THIRTY_AND_ONE), True)
## Test 4 for earlier - second earlier than first, differ in seconds
check.expect("Q4-e-4", earlier(EIGHT_THIRTY_AND_ONE, EIGHT_THIRTY), False)
## Test 5 for earlier - same time
check.expect("Q4-e-5", earlier(EIGHT_THIRTY_AND_ONE, EIGHT_THIRTY_AND_ONE), False)
```

## 5.3 A code interface

Here is an example of a code interface, such as might be provided on an assignment. The docstring for the class definition has been omitted; in assignments, you will need to fill in the details.

```
class Multiset:

    def __init__(self):
        """
        Multiset() produces a newly constructed empty Multiset.
        Effects: Creates a new empty Multiset.
        __init__:  -> Multiset

        """

    def __contains__(self, value):
        """
        value in self produces True if value is an item in self.
        __contains__: Multiset Any -> Bool

        """

    def add(self, value):
        """
        self.add(value) adds value to self.
        Effects: Mutates self by adding value to self.
        add: Multiset Any -> None

        """

    def delete(self, value):
        """
        self.delete(value) removes an item with value from self.
        Effects: Mutates self by removing an item with value from self.
        delete: Multiset Any -> None
        Requires: self contains an item with value value

        """
```

As an alternative to the docstring, comments can be used before each method, as shown in the example below:

```
class Multiset:

    ## Multiset() produces a newly constructed empty multiset.
    ## Effects: Creates a new empty Multiset.
    ## __init__:  -> Multiset
    def __init__(self):

    ## value in self produces True if value is an item in self.
    ## __contains__: Multiset Any -> Bool
    def __contains__(self, value):

    ## self.add(value) adds value to self.
    ## Effects: Mutates self by adding value to self.
    ## add: Multiset Any -> None
    def add(self, value):

    ## self.delete(value) removes an item with value from self.
    ## Effects: Mutates self by removing an item with value from self.
    ## delete: Multiset Any -> None
    ## Requires: self contains an item with value value
    def delete(self, value):
```

## 5.4 A testing file

To test the file `pair.py`, this testing file imports both `check.py` and `pair.py`.

The file creates objects of type Pair, and then uses them to test the various methods for the class.

```
import check
from pair import *

one = Pair("one", 1)
two = Pair("two", 2)
three = Pair("junk", 0)

check.expect("Testing first", one.pair_first(), "one")
check.expect("Testing second", one.pair_second(), 1)

check.expect("Testing repr", repr(one), "Pair(First = one, Second = 1)")
three.add_first("three")

check.expect("Testing add_first", three.pair_first(), "three")
three.add_second(3)
check.expect("Testing add_second", three.pair_second(), 3)

check.expect("Testing contains, first", "one" in one, True)
```

```
check.expect("Testing contains, second", 1 in one, True)
check.expect("Testing contains, neither", "cat" in one, False)

check.expect("Testing equal with same Pair", one == one, True)
check.expect("Testing equal with different Pairs", one == Pair("one", 1), True)
check.expect("Testing not equal", one == two, False)

check.expect("Testing that one is a Pair", isinstance(one, Pair), True)
```