Style Guide for CS 231 and CS 234

Naomi Nishimura

December 11, 2019

1 Introduction

This document specifies the style required for Python programs written for the course. It assumes familiarity with the design recipe, as discussed in detail in CS 115, CS 116, CS 135, and CS 136, and uses standards set out for Python in PEP 8 – Style Guide for Python Code (`legacy.python.org/dev/peps/pep-0008`).

You may find discrepancies among the styles used in the textbook, 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), detailed instructions for testing (Section 4), guidelines for definitions of user-designed types (Section 5), and sample submissions that satisfy style requirements (Section 6).

2 Assignment format

Each assignment should start with an assignment header, include comments, and make use of various techniques for enhancing readability.

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 2018
##   Assignment 03, Problem 4
## ======================================================
##
```

The remainder of the file should appear in the following order, with documentation for each function (discussed in detail in Section 3) appearing before the function definition: assignment header, import statements (if any), definitions of user-defined types (if any), constant definitions, helper functions, main program/-function.

User-defined types (especially classes) are particularly important in CS 234; be sure to read Section 5 carefully to ensure that you follow the proper format.

2.2 Comments

Most of the comments you provide will appear in a block at the beginning of a function, as dictated by the design recipe. Part of the marks for each assignment will be allocated for such comments.

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 function, a function definition, or a class definition. An alternative way of separating information is the use of a row of =’s or other symbols, as used in the assignment header.

If the 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 with the function they are helping. Typically, helper functions are placed before the main function, though at times other locations may be acceptable: remember that the goal is to make it easy for the reader to determine where your functions are located.

The use of indentation of 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.

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.

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

- To improve the readability of your code by avoiding "magic" numbers.
- To improve flexibility and allow easier updating of special values.
- To 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 (also illustrated in the examples given in Section 6):

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

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 [comments]

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.

For example, this 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.

```
## 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 [comments]

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 [comments]

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 [comments]

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, 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 example should you write the function body. You should also write class definitions (discussed in Section 5) and helper functions before main functions.

3.7 Tests [code]

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.

Details about tests can be found in the next section.

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

4 Tools for testing

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 website, 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 assignment.

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 Test 6"), 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 1 Test 6"), 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.

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 seven steps, where only the relevant steps are required in any particular test; at a minimum, each test will contains steps 1 and 6.

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 user input from the keyboard (if any).
If your function uses keyboard input (such as `input`), use `check.set_input` before running `check.expect` or `check.within`. The function `check.set_input` consumes a list of strings, which will be used as input for `input` instead of requiring you to type in values to `input` when you run your `check.expect` or `check.within`. If the list you provide to `check.set_input` is the wrong length, an error will occur.

Step 4: Check printing to the screen (if any).
If you wish to test a call to a function with screen output, use `check.set_screen` before running `check.expect` or `check.within`. The function `check.set_screen` consumes a string describing what you expect your function to print. When you call `check.set_screen`, the next use of `check.expect` or `check.within` will print both the output of the function you are testing and the expected output you gave to `check.set_screen`.
Since screen output will not effect whether `check.expect` or `check.within` passes or fails, you need to visually compare the output to make sure it is correct. You can choose the format of the string passed to `check.set_screen` in a way that makes the visual comparison easy for you.

Step 5: Check writing to files (if any).
If your function writes to a file, you will need to use either `check.set_file` or `check.set_file_exact` before running `check.expect` or `check.within`. Each of these functions consumes two strings: the first is the name of the file that will be produced by the function call in step 6, and the second is the name of a file identical to the one you expect to be produced 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. If the files are the same, the test will print nothing; if they differ, the test will print which lines don't match, and will print the first pair of differing lines for you to compare. If you use `check.set_file`, the two files are compared with all white space removed from both files. If you use `check.set_file_exact`, the files must be character-for-character identical in order to be considered the same.

Step 6: 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 7: 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 field (classes), and for each possibility (mixed types)

5 User-defined data

The ideas used in forming contracts and **Requires** sections are also used in specifying user-defined data types.

5.1 Class definitions

CS 234 makes extensive use of code interfaces in the form of class definitions. A sample can be found in Section 6.5.

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 fields and their types; if there are further restrictions on the fields, add a **Requires** section, as in the sample given in Section 6.4.

Each method should be documented like a function.

5.2 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"
```

6 Sample Python programs

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 such as, for example, a multi-line purpose statement.

All files can be found on the Python page of the course website, including extra files with errors in the tests so that you can see what type of information is provided.

6.1 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 2018                        *
## *    Assignment 03, Problem 1                  *
## ************************************************
##

import check

# 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].

def add_one_to_evens(int_list):
    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])
```

6.2 Using keyboard input and screen output

Notice the use of `check.within` for the second and third tests, since division results in a floating point number.

In the fourth test, `check.set_input` is used to simulate a user typing `hello` as keyboard input, and `check.set_screen` is used for the expected screen output. Since the output is checked visually against the actual screen output, adding the prompt `Enter a non-empty string:` is not required.

Bad tests (incorrect output and wrong length of input list) can be found in `ex2witherrors.py`, linked off the Python page of the course website.

```
##
## !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
##
##   Ima Student (12345678)
##   CS 231 Spring 2018
##   Assignment 03, Problem 2
##
## !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
##
##

import check

## mixed_fn(num, action) produces 2*num if action is "double",
##     num/2 if action is "half", and None otherwise.
##     If action is "string", the user is prompted to enter a string
##     and then num copies of that string are printed on one line.
##     For any other action, "Invalid action" is printed to the screen.
## Effects: If action is "string", a string is read from standard input,
##     and copies of the string are printed. For any action other than
##     "string", "half", or "double", "Invalid action" is printed.
## mixed_fn: Int Str -> (anyof Int Float None)
## Requires: num is non-negative
## Examples:
##     mixed_fn(2, "double") => 4
##     mixed_fn(11, "half") => 5.5
##     mixed_fn(6, "oops") => None and prints "Invalid action"
##     mixed_fn(3, "string") => None, if the user inputs "a"
##         then "aaa" is printed and nothing is produced

def mixed_fn(num,action):
    if action == "double":
        return 2 * num
    elif action == "half":
        return num / 2
    elif action == "string":
        to_print = input("Enter a non-empty string: ")
        print(to_print * num)
    else:
```

```
        print("Invalid action")

# Test 1: action == "double"
check.expect("Q2T1", mixed_fn(2, "double"), 4)

# Test 2: action == "half", odd number
check.within("Q2T2", mixed_fn(11, "half"), 5.5, .001)

# Test 3: action == "half", even number
check.within("Q2T3", mixed_fn(20, "half"), 10.0, .001)

# Test 4: action == "string"
check.set_input(["hello"])
check.set_screen("hellohellohello")
check.expect("Q2T4", mixed_fn(3, "string"), None)

# Test 5: invalid action
check.set_screen("Invalid action")
check.expect("Q2T5", mixed_fn(2, "DOUBLE"), None)
```

6.3 Using file input and output

In this example, the function `check.set_file` is used to select a file representing the expected contents of the output file `summary.txt`. All relevant files can be found on the Python page on the course website, as well as the file `ex3witherrors.py`, which demonstrates what happens when files do not match when compared during Step 5 of testing. When you run the file with errors, `PASSED` is printed even for the bad test, since all that is required is that the value produced matches the expected value.

```
##
##   ////////////////////////////////////////
##   ///    Ima Student (12345678)          ///
##   ///    CS 231 Spring 2018              ///
##   ///    Assignment 03, Problem 3        ///
##   ////////////////////////////////////////
##

import check

## 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 from the file called fname
##          Writes to the file called "summary.txt"
## file_filter: String Int -> None
## Requires: 0 <= minimum <= 100
##           fname is the name of a readable file
```

```
## Examples:
##     If "ex1.txt" is empty, then file_filter("ex1.txt", 1)
##         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)

def file_filter(fname, minimum):
    infile = open(fname, "r")
    input_list = infile.readlines()
    infile.close()
    outfile = open("summary.txt", "w")
    for line in input_list:
        if int(line.strip()) > minimum:
            outfile.write(line)
    outfile.close()

# Test 1: empty file (example 1)
check.set_file("summary.txt", "empty.txt")
check.expect("Q3T1", file_filter("empty.txt", 40), None)

# Test 2: example 2
check.set_file("summary.txt", "eg2-summary.txt")
check.expect("Q3T2", file_filter("eg2.txt", 50), None)

# Test 3: file contains one value, it is > minimum
check.set_file("summary.txt", "one-value.txt")
check.expect("Q3T3", file_filter("one-value.txt", 20), None)

# Test 4: file contains one value, it is < minimum
check.set_file("summary.txt", "empty.txt")
check.expect("Q3T4", file_filter("one-value.txt", 80), None)

# Test 5: file contains one value, it equals minimum
check.set_file("summary.txt", "empty.txt")
check.expect("Q3T5", file_filter("one-value.txt", 50), None)

# Test 6: file contains 1-30 on separate lines
check.set_file("summary.txt", "sixteen-thirty.txt")
check.expect("Q3T6", file_filter("thirty.txt", 15), None)
```

6.4 Using a class definition

```
##
## @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
##   Ima Student (12345678)
##   CS 231 Spring 2018
##   Assignment 03, Problem 4
## @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
##
```

```
import check

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

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

    ## print(self) produces a string with hour, minute, and seconds.
    ## __str__: Time -> Str
    def __str__(self):
        return "{0:.2}:{1:.2}:{2:.2}".format(self.hour, self.minute, self.second)

    ## self == other produces True if self and other are equal and False otherwise
    ## __eq__: Time Any -> Bool
    def __eq__(self, other):
        return instance(other, time) and \
               self.hour == other.hour and \
               self.minute == other.minute and \
               self.second == other.second


## 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)

## 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

def time_to_seconds(t):
```

```
    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)


## 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

def earlier(time1, time2):
    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)
```

6.5 A code interface

```
class Multiset:

    ## Multiset() produces a newly constructed 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):
```