The Design Recipe for Python

You will submit your Python assignments in the same way as you submitted your Scheme assignments; just be sure to use the suffix `.py`, as in `a06q2.py`. When you use the WingIDE, files are automatically saved with the `.py` suffix.

Contract: We will continue to use the same structure for describing the contracts of any functions that we write. However, there are a few changes to be aware of in moving from Scheme to Python. In the table below, rows marked with a C are changes from Scheme, and rows marked with an N are new data types for Python introduced in modules 8 and 9.

C	`str`	for strings
	`int`	for integers
C	`int[>=0]`	for natural numbers
C	`float`	for floating point (non-integer) numbers
C	`bool`	for Boolean values (`True` and `False`)
	`(list <t1>...<tn>)`	for a list of exactly n elements, where the first element has type `<t1>`, the second has type `<t2>`, etc.
	`(listof <type>)`	for a list of arbitrary length with elements of type `<type>` (could be `int`, `str`, a `union`, etc.)
C	`None`	for functions that consume nothing and/or produce nothing, in the same way that `(void)` was used in Scheme.
	`any`	for values of unknown type
	`X, Y, etc.`	for values of unknown type, but must match. For example, use `X` if a function consumes two values where the only restriction is that the two values must have the same type
	`<other>`	for other types; any type that is defined with a data definition in the question document or in a comment in your code can be used as a type in your contracts.
	`union`	for mixed data types. For example, `(union int float)` should be used to denote any number.
N	`(dictof <t1> <t2>)`	for dictionaries, where `<t1>` is the key type and `<t2>` is the value type. For example, the dictionary wavelengths in Module 8 is of the type `(dictof str int)`.
N	`<class name>`	for classes. For example, `Country` in Module 8.
N	`file`	for a file. Keep in mind that most functions on assignments will consume a filename (a string), not an actual file.
N	`(tuple <t1>...<tn>)` `(tupleof <type>)`	for tuples; use in the same way as `list` and `listof`

Common Errors: The following two points cause a lot of students to loose marks on assignments:

- Do NOT use `num` or `number`; the correct type is `(union int float)`
- Data collected through `raw_input` is NOT consumed by the function
- Anything printed to the screen is NOT produced by the function.

Purpose: As before, you should mention the parameters consumed by the function, as well the value produced (if any), and you should explain how the consumed information relates to the produced information (using parameter names). The purpose should only mention values consumed by the function and produced (using a `return` statement). Anything else (such as `print` statements, mutation, `raw_input`, file I/O, etc.) is a side effect, and should be described in the "**Effects**" section, as discussed for Advanced Scheme.

Effects: We will often write functions that in addition to, or instead of, producing a value, have side effects. These side effects should be explained as well. As with the purpose, try to use parameter names to explain the effects as much as possible.

Examples: Again, as before, you should include a few examples to illustrate what your function does. If there are no side effects, it is sufficient to indicate the value produced when the function is called with specific values, as done previously. However, when there are side effects, these need to be described in English.

Body: Complete the code in Python, using helper functions as needed; note that helper functions should be declared outside the function body. As in Scheme, you should include contract, purpose, and effects for helper functions, but you do not need to include examples or testing.

Our Scheme program bodies were generally short enough that our design recipe 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 a marker can understand your intentions more clearly. In particular, comments can be useful to explain the role of local variables, if statements, and loops.

Testing: Python doesn't present us with a function like `check-expect` in Scheme for testing our programs; in order to emulate this functionality, you can download the module `check.py` 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 correct cases to test rather than how to test your code. In order to use it, you must save `check.py` in the same directory as your `aXqY.py` files, and include the line `import check` at the beginning of each Python file. You do not need to submit `check.py` when you submit your assignment.

Our tests for most functions will consist of up to five parts; you only need to include the parts that are relevant to the function you are testing (for example, you can skip steps 1 and 6 if there are no state variables). More detail on these parts can be found following the summary.

1. Write a brief description of the test as a comment
2. If there are any state variables to consider, set them to specific values.
3. If you expect user input from the keyboard, call `check.set_input(lst)`.
4. If you expect your function to print anything to the screen, call `check.set_screen(s)`.
5. If your function writes to any files, call `check.set_files`.
6. Call either `check.expect` or `check.within` with a your function as the value to test.
7. If the effects of the function include mutating state variables, call `check.expect` or `check.within` once for each state variable, with that variable as the value to test.

Additional information about testing:

Step 1: If your function reads from a file, you will need to create the file (using a text editor like Wing IDE) 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 description of what is contained in the files read in that test.

Step 2: You should always set the values of every state variable on every test, in case your function inadvertently mutates their values.

Step 3: If the value to test is a call to a function with screen output, you need to use `check.set_screen` (which consumes a string describing what you expect your function to print) before running the test. **Screen output has 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.

Step 4: If your function uses keyboard input (such as `raw_input`), you will need to use `check.set_input` before running the test. This function consumes a list of strings, which it will use as input the next time you run your function. This way you do not need to do any typing when you run your tests. You will get an error if the list you provide to `check.set_input` is the wrong length.

Step 5: If your function writes to a file, you will need to use either `check.set_file` or `check.set_file_exact` before running the test. 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 the first pair of differing lines for you to compare. If you use `check.set_file`, the two files will are "the same" even if the whitespace is different; if you use `check.set_file_exact`, the files must be character for character identical in order to be considered "the same."

Steps 6 and 7: 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; 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"), 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 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. 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 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.

Note that `check.expect` and `check.within` only print a message if your test fails: if nothing gets printed, the test passed.

Examples of Code Following the Design Recipe

(Mutation)

```
# add_one_to_evens: (listof int) -> None
# Purpose: Consumes a list, lst, and produces nothing.
# Effects: Mutates lst so that all even integers in lst
#          are increased by 1.
# Examples: if L = [0,1,2,3,4,5] and add_1_to_evens(L)
#              is called, then L = [1,1,3,3,5,5].
#           if L = [3,5], and add_one_to_evens(L) is
#              called, then L = [3,5] afterwards.

def add_one_to_evens(lst):
    for i in range(len(lst)):
        if lst[i]%2==0:
            lst[i] = lst[i] + 1

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

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

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

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

(Keyboard Input and Screen Output)

```
# mixed_fn: int str -> (union int float None)
# Purpose: consumes an integer n and a string action and produces
#     a value according to the following rules:
#   * If action is "double", produces the integer 2*n
#   * If action is "half", produces the float 0.5*n
#   * If action is "string", produces None
#   * If action is any other value, produces None.
# Effects:
#   * If action is "double" or "half", no effects
#   * If action is "string", then the user is prompted to enter
#     a string, that string is then concatenated to itself n times
#     and printed to the screen
#   * Otherwise, prints "Invalid action" to the screen.
# Examples:
#     mixed_fn(2,"double") => 4
#     mixed_fn(11, "half") => 5.5
#     mixed_fn(6,"oops") produces None and prints "Invalid action"
#     mixed_fn(3,"string") will prompt the user for a string; if the
#       user inputs "a" then "aaaaa" is printed and nothing is produced

def mixed_fn(n,action):
    if action=="double":
        return 2*n
    elif action=="half":
        return n/2.0
    elif action=="string":
        s = raw_input("enter a non-empty string: ")
        print s*n
    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"), 3.5, .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)
```

(File Input and Output)

```
# file_filter: str int[>=0, <=100] -> None
# Purpose: consumes string fname, representing a filename,
#          and an integer, minimum, between 0 and 100
# Effects: Reads integers (one per line) from the file with
#          name fname, and writes each of those integers
#          which is greater than minimum to a new file, summary.txt
# 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("ex1.txt", 50) will create a file
#      named summary.txt containing 75, 90 (one per line)

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

# Test 1: empty file
# q3t1_input.txt contains nothing
check.set_file("summary.txt", "q3t1_expected.txt")
check.expect("Q3T1", file_filter("q3t1_input.txt", 40), None)

# Test 2: one integer smaller than minimum
# q3t2_input.txt contains 12
check.set_file("summary.txt", "q3t2_expected.txt")
check.expect("Q3T2", file_filter("q3t2_input.txt", 40), None)

# Test 3: one integer larger than minimum
# q3t3_input.txt contains 76
check.set_file("summary.txt", "q3t3_expected.txt")
check.expect("Q3T3", file_filter("q3t3_input.txt", 40), None)

# Test 2: general case
# q3t4_input.txt contains thirty integers, equally split
#    above and below 65
check.set_file("summary.txt", "q3t4_expected.txt")
check.expect("Q3T4", file_filter("q3t4_input.txt", 65), None)
```