The Design Recipe for Python

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.

- Use the types recognized by Python: `int` for integers, `float` for floating point numbers, `str` for strings, `bool` for booleans. Do not use *nat* and *num* – they are not Python types. You can also names of classes as types (such as `Country` in Module 8).
- If a Python function consumes or produces a number that can be either an integer or a floating point number, the correct type is `(union float int)`. You will need to write your function carefully to take into account the differences in the way `float`s and `int`s are handled by arithmetic operations in Python.
- Use `None` to the left of the arrow if the function does not consume any values, and use `None` to the right of the arrow if the function does not return any value.
- Continue to use *union* for mixed types, *listof* for lists, and square brackets for adding constraints (such as *(listof* `str`*)[nonempty] and* `int` *[>=0]*).
- Use *(dictof type1 type2)* for dictionaries. For example, the dictionary `wavelengths` in of Module 8 is of the type *(dictof* `str int`*)*.
- Use `file` for a file.
- Tuples are used in Modules 9 and 10 only. Use *(tupleof type1 type2 ... typeN)* for tuples. For example, the `tuple(1,True)` is of the type *(tupleof* `int bool`*)*.
- Continue to use *X*'s and *Y*'s to indicate the relationship between types, as we have seen in contracts for functions. For example, the type of the `items` method for dictionaries in Module 10 is:

```
##  dict.items: (dictof X Y) -> (listof (tupleof X Y))
```

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). If a function has side effects, then describe them in a separate "**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 (including reading and writing information provided by the user as the program is running, or mutating variables or parameters). 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 several 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 (mutations as well as input/output), these should be described. It is often easier to write this in English, rather than trying to write it in Python.

Body: Complete the code in Python, using helper functions as needed. Note that helper functions should be declared outside the function body, and you should follow the full design recipe for them, including 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. 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. Some places where comments may prove useful include:

- to explain the role of non-obvious local variables,
- at the beginning of blocks of code, such as if statements and loops, to indicate what the block is doing,

Testing: Python doesn't present us with a function like `check-expect` in Scheme for testing our programs. We will have to do our testing differently. How we test will depend on the type of function being tested. As before, your tests will be marked by hand, and not actually run. As usual, the tests must follow the definition of the function.

Our tests will still consist of three parts: setting up a known situation (setting variables and values for our function call), calling the function, checking the expected result with the actual result observed. Sometimes these steps can be combined.

- Functions which produce values – these tests are most like what we have done before, as we need to check the actual and expected values for equality. In Scheme, we would use `check-expect`. In Python, we can take a couple of different approaches, both of which are suitable to submit for assignments.
  - Use a print statement to print out the result of comparing the expected and actual values using the equality test `==`. You will know the test has passed by observing the value `True` printed, and that it has failed when you see `False`.
  - Note that when the values you wish to compare are floating point numbers, the `==` test for equality may not be sufficient. Instead, you'll need to determine if the values are "close enough". This can be done by writing your own testing helper function to check the absolute value of the difference between the expected and actual values. Please see the use of the function `close` in the examples below.
  - Use the Python statement `assert`, followed by the test for equality of the expected and actual values. If the assertion passes (i.e. the test evaluates to `True`), then the program runs without complication. If the assertion fails, then your program crashes with an `AssertionError`.
  - See `mixed_fn` tests 1-3 for examples of the expected format.

- Functions with mutation side effects
  - The testing of these functions is similar to testing functions which return values. The difference is that you do not test the actual value returned by the function to the expected value, but rather you test the new value of the variable (or component of the value) with the value you expected it to have once the function has changed it. The actual testing can be done using print tests for equality and/or `assert`. You compare the expected value of a variable (or list component or field of an object) with the actual value it has after the function has executed.
  - See the tests for `add_one_to_evens` for expected format.

- Functions with reading and printing side effects (standard input and output)
  - If a function includes a `raw_input` comment, this is difficult to document. You can simply include a comment or a print statement indicating what should be entered, and then indicate what should happen (either as returned values or side effects).

- If a function includes print statements, there are limited ways to test those for equality. Basically, you can print (or just describe with a comment) the output you would expect to see for the input provided (that input can be via parameters or from the keyboard or a file)
- See `mixed_fn` tests 4,5 for expected format for this type of tests.

- Function which involve files for input or output
  - Actual testing of this type of function involves creating multiple input files, covering various situations to be considered (e.g. empty file, file contains duplicate information, etc.). For assignments, it will be sufficient to provide a description of what the file would contain, rather than submitting the actual file, unless otherwise indicated on the assignment.

- Functions which return values and have mutation or printing side-effects
  - You should use a combination of the testing described.

On your assignments, please use the following format for your tests:

```
## describe the test, using => notation if appropriate
print "Test n"                        ## where n is 1,2,3, …
Python code to check the result      ## use equality testing and/or
                                      ## assert where possible
```

Two annotated examples follow:

```
## mixed_fn: int string -> (union int float None)
## purpose: consumes an integer n and a string action and produces a number
## 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", prompts the user to enter a string, s and
##   then prints out the string s*n (i.e. s concatenated to itself n times)
## * if action is any other value, produces None.
## effects:
## * if action is "string", then the user is prompted to enter a string,
## that string is then concatenated to itself n times and displayed
## * if action is anything other than "double","half", or "string" then
## the message "Invalid action" is displayed.
## example:
## mixed_fn(2,"double") => 4
## mixed_fn(11, "half") => 5.5
## for the call mixed_fn(6,"oops"), no value is returned but the message
## "Invalid action" is displayed.
## for the call mixed_fn(3,"string"), if the user enters "hi!", then
## "hi!hi!hi!" is displayed.
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"

## Testing
## mixed_fn(2, double) => 4
print "Test 1"
expected = 4
ans = mixed_fn(2,"double")
print ans==expected
## in addition (or instead of) the above print, could use assert
assert ans==expected


## Helper function used for testing functions involving
## floating point numbers
## close: (union float int) (union float int) -> bool
## consumes two numbers and produces True if their absolute value
## differs by less than 0.000001, False otherwise.
## examples: close(1.2345674, 1.2345681) => True
##           close(1.234, 1.2345) => False
def close(x,y):
    return abs(x-y) < 0.000001

## mixed_fn(11, "half") => 5.5
print "Test 2"
expected = 5.5
actual = mixed_fn(11,"half")
print close(actual, expected)
assert close(actual, expected)

## mixed_fn(10, "half") => 5.0
print "Test 3"
expected = 5.0
actual = mixed_fn(10,"half")
print close(actual, expected)
assert close(actual, expected)

## mixed_fn(3, "string") => None, displays input string three times
print "Test 4"
print "If user enters 'hi!', then 'hi!hi!hi!' printed"
mixed_fn(3, "string")

## mixed_fn(100, "oops") displays "Invalid action"
print "Test 5"
print "Expect to see 'Invalid action' printed"
print "Actual output follows"
mixed_fn(100, "oops")


## add_one_to_evens: (listof int) -> None
## effects: mutates L so that all even integers in L
## 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(L):
    for ind in range(len(L)):
        if L[ind]%2==0:
            L[ind] = L[ind] + 1

## Testing
## empty list should be unchanged
print "Test 1: empty list"
L = []
add_one_to_evens(L)
print L==[]

## len(L)=1, value even => only list value increased by 1
print "Test 2: length 1, even number"
L = [2]
add_one_to_evens(L)
print L==[3]

## len(L)=1, value odd => list unchanged
print "Test 3: length 1, odd number"
L = [3]
add_one_to_evens(L)
print L==[3]

## len(L)>1, even/odd/0/neg values => list mutated
print "Test 4: length > 1"
L = [3,4,0,2,1,8,-4]
add_one_to_evens(L)
print L==[3,5,1,3,1,9,-3]
```

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 `a6q2.py`. When you use the WingIDE, files are automatically saved with the `.py` suffix.