Computer Science @ The University of Waterloo

CS 116
Course Specific Style Guide

Last Updated: 2019.10.15

1 Introduction

In addition to the **common style guide** for CS courses at the University of Waterloo, this document will help explain the course specific changes needed for CS 116. This document will also include specific examples outlining how the style guide should look for this course.

2 Assignment Formatting Changes

With Python, functions that we write can now do more than simply produce a value. With mutation, a function can change the value of list, dictionary, or object arguments, or have other effects. While allowing programs to do even more, the introduction of mutation requires changes to some steps of our design recipe, along with the introduction of a new step. Students should first read the **common first year style guide** before this document.

2.1 Whitespace

As Python is an indentation language, whitespace will largely be forced by the structure of the language. Be consistent and either use two spaces for indentation or 4 spaces throughout all of your assignments. Do not mix tab characters and spaces as it will cause errors when running in Python. Always use spaces (check your IDE for instructions as to how to make your tab character type in spaces instead of tab characters). Also note that `import` calls should appear one line after the header and should be listed in alphabetical order. Two blank lines should follow these calls.

2.2 Naming Conventions

- The dash ("-") character cannot be used as an identifier in Python. You should instead use an underscore ("_") where a dash would have been used (e.g. `tax_rate`).
- Variables should begin with lower case letters (variables that represent lists can be a single capital letter) and should be written in either snake case or camel case.

2.3 Comments

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

2.4 Function Headers

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

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

2.5 Python's Docstring

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

we will use three single quotes to signify the beginning and end of our docstring for a function.

2.6 Helper and Wrapper Functions

Helper functions may be defined locally in your Python programs, but it is not required nor recommended. Tests and examples are not required for any functions you are not explicitly told to write by a question in CS 116 (eg. helper functions). They are always required for functions you are asked to write unless a question dictates otherwise.

2.7 Purpose

As before, your purpose statements should briefly explain what the function does using parameter names to show the relationship between the input and the function's actions.

There are two small changes when writing our purpose statements:

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

2.8 Contract

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

- Python does not have a symbol type. You should use strings or numbers in the role of symbolic constants as needed. (Python does have an enumerated type, but it is not used in CS 116).
- Python does not have a character type. Instead, use strings of length one.
- Python does not use the `Num` type. If a value can be either an integer or non-integer value, use `(anyof Int Float)`.

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

The following table lists the valid Python types:

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

2.9 Examples

Unlike in Racket, examples in Python cannot be written as code using the provided `check` module. Unfortunately, the function calls in the `check` functions cannot come before the actual function definitions. Therefore, instead of writing examples as code, we will include them as part of our function's docstring. The notation to write an example is to use `fcn_call(input_data) => output`. The format of the example depends on whether or not the function has any effects. Examples that fit on one line can be written as so. Otherwise the first sentence should start on a new line and indented 2 or 3 spaces (be consistent).

If the function produces a value, then the example can be written as a function call, with its expected value.

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

2.10 Testing

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

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

1. Write a brief description of the test as the descriptive label in the testing function or as a comment
2. If there are any global state variables to use in the test, set them to specific values.
3. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`). `check` function after the return value test, once for each mutated value to be checked.

Additional information about testing:

- You should always set the values of every global state variable in every test before calling any `check` functions, in case your function inadvertently mutates their values.
- The two main functions included in `check` are `check.expect` and `check.within`; these functions will handle the actual testing of your code. You should only use `check.within` if you expect your code to produce a floating point number or an object containing a floating point number. In every other case, you should use `check.expect`. When testing a function that produces nothing, you should use `check.expect` with `None` as the expected value.
  - `check.expect` consumes three values: a string (a label for the test, such as "Question 1 Test 6" or a description of the test case), a value to test, and an expected value. You will pass the test if the value to test equals the expected value; otherwise, it will print a message that includes both the value to test and the expected value, so that you can see the difference.
  - `check.within` consumes four values: a string (a label such as "Question 1 Test 6", or a description of the test), a value to test, an expected value, and a tolerance. You will pass the test if the value to test and the expected value are close to each other (to be specific, if the absolute value of their difference is less than or equal to the tolerance); otherwise, it will print a message that includes both the value to test and the expected value, so that you can compare the results.
- **Note:** Examples should be rephrased as tests in Python since the `check` module does not read the docstring for tests (see the exemplars below).

2.11 Final Thoughts

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

3 Module 1

3.1 Module 1: Basic Example with Check Expect

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 1 Sample
## ************************************************
##

import check
import math


##Constants go here
lucky_seven = 7

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

    add_7: Int -> Int

    Examples:
       add_7(0) => 7
       add_7(-7) => 0
       add_7(100) => 107
    '''
    return n + lucky_seven

##Examples:

check.expect("Testing first example", add_7(0), 7)
check.expect("Testing second example", add_7(-7), 0)
check.expect("Testing third example", add_7(100), 107)

##Tests:

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

3.2 Module 1: Basic Example with Check Within

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 1 Sample
## ************************************************
##

import check
import math


##Here is where constants would go if necessary

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

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

    Examples:
       circle_area(0.0) => 0.0
       circle_area(1.0) => 3.141592653589
    '''
    return math.pi*r*r

##Examples:

check.within("Example 1: Zero area", circle_area(0.0), 0.0, 0.00001)
check.within("Example 2: Pi", circle_area(1.0), 3.1415926, 0.00001)

##Tests:

check.within("small value", circle_area(0.01), 0.0003141592, 0.00001)
check.within("large value", circle_area(100), 31415.926535, 0.00001)
```

3.2.1 Module 1: Return Strings

```
##
## ************************************************
##   Rita Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 1 Sample
## ************************************************
##

import check


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

    cat: Str Str -> Str

    Examples:
       cat("", "") => ""
       cat("top", "hat") => "tophat"
    '''
    return s + t

##Examples

check.expect("Testing first example", cat("", ""), "")

check.expect("Testing second example", cat("top", "hat"), "tophat")

##Tests

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

4 Module 2

4.1 Module 2: Recursive Example

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 2 Sample
## ************************************************
##

import check


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

    factorial: Nat -> Nat

    Examples:
       factorial(0) => 1
       factorial(1) => 1
       factorial(5) => 120
    '''
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

##Examples:

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

##Tests:

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

5 Module 3

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

5.1 Purpose

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

5.2 Effects

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

- Printing to screen
- Taking input from user

Effects should be written **after** the purpose statement but **before** the contract. A white space should precede and follow the effects section.

5.3 Examples

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

- 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() => None
     and the following is written to the screen:
  Waterloo, Ontario
  '''
  ```

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

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

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

5.4 Testing

We need to make some additions to testing as well to account for side effects:

1. Write a brief description of the test as the descriptive label in the testing function or as a comment
2. If there are any global state variables to use in the test, set them to specific values.
3. If you expect user input from the keyboard, call `check.set_input`.
4. If you expect your function to print anything to the screen, call `check.set_screen` or `check.set_print_exact`.
5. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`). `check` function after the return value test, once for each mutated value to be checked.

Additional Notes:

- Step 3: If the value to test is a call to a function that prints to the screen, you have two choices for checking the printing: `check.set_screen` and `check.set_print_exact`.

  You can use `check.set_print_exact` (which consumes one string for each line you expect your function to print to the screen) before running the test. When the test is run, in addition to comparing the actual and expected returned values, the strings passed to `check.set_print_exact` are compared to what is actually printed by your function call. Two messages will be printed: one for the returned value and one regarding the printed strings.

Important: This command will not test for output printed to the screen from a `input` call.

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

- Step 4: If your function uses keyboard input, you will need to use the command `check.set_input` before running the test. This function consumes strings corresponding to the input that will be used instead of waiting for data to be typed in when the function is called. You do not need to do any typing for `input` when you run your tests. You will get an error if you do not have enough strings in your call to `check.set_input` and your test will fail if you don't need all the provided strings. You must have exactly the correct number of strings.

5.4.1 Module 3: Printing Example

Notice below you can choose to include that the function returns `None` or omit this since the contract captures this information. Try to use only one of `set_print_exact` or `set_screen` in your testing.

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 3 Sample
## ************************************************
##

import check


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

    Effects: Prints to the screen

    print_it_three_times: Str -> None
```

```
    Examples:
       print_it_three_times("") => None
       and prints three blank lines
       print_it_three_times("a") => None
       and prints a once on each of three lines
    '''
    print(s)
    print(s)
    print(s)

##Examples:

check.set_print_exact("", "", "")
check.expect("Example 1: Empty string", print_it_three_times(""), None)
check.set_print_exact("a", "a", "a")
check.expect("Example 2:  Single character", print_it_three_times("a"), None)

##Examples using set_screen:

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

##Tests:

check.set_print_exact("CS 116", "CS 116", "CS 116")
check.expect("Test random string", print_it_three_times("CS 116"), None)

##Tests using set_screen:

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

5.4.2 Module 3: Multiple Keyboard Input

```
## ************************************************
##   Rita Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 3 Sample
## ************************************************

import check


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

    Effects:
       Reads input from keyboard
       Prints to screen

    diff: None -> None

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

       If the user enters enters 0 and 0 after
       diff() => None
       is called, then 0 is printed to the screen
    '''
    x = int(input("Enter an integer"))
    y = int(input("Enter another integer"))
    return x - y

##Examples

check.set_input ("5", "3")
check.set_print_exact("2")
check.expect("Testing first example", diff(), None)

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

##Tests

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

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

5.4.3 Module 3: Keyboard Input and Screen Output

```
##
## ************************************************
##   Rita Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 3 Sample
## ************************************************
##

import check


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

    Effects:
       If action is "string":
          Reads input form keyboard
          Prints to Screen
       If action is not "string", "half" or "double":
          Prints to screen

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

    Examples:
       mixed_fn(2, "double") => 4
       mixed_fn(11, "half") => 5.5
       mixed_fn(6, "oops")  => None
         and prints "Invalid action"
       If the user inputs "a" after calling
         mixed_fn(5, "string") => None
         and "aaaaa" is printed
    '''
    if action=="double":
        return 2*n
    elif action=="half":
        return n/2
    elif action=="string":
        s = input("enter a non-empty string: ")
        print (s*n)
    else:
        print ("Invalid action")

##Examples

check.expect("Try double", mixed_fn(2, "double"), 4)
```

```
check.within("Try half with odd", mixed_fn(11, "half"), 5.5, 0.0001)

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

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

##Tests

check.within("Try half with even", mixed_fn(20, "half"), 10.0, 0.0001)

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

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

##A test using set_screen:

check.set_input ("word")
check.set_screen("word repeated 10 times without spaces")
check.expect("try string", mixed_fn(10, "string"), None)
```

6 Module 4

In module 4, we introduce lists, mutable objects and mutation for the first time. We need to update our effects and testing sections accordingly.

6.1 Purpose

Actions functions perform may include producing a value, mutations and the use of `input` and `print`.

6.2 Effects

Our effects now can include:

- Printing to screen
- Taking input from user
- Mutating a parameter (eg. a list or in later modules, a dictionary or class object)

6.3 Examples

We also need to make additions to examples to account for mutations:

If the function involves mutation and returns `None`, the example needs to reflect what is true before and after the function is called.

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

6.4 Testing

We now include mutation in our testing routine:

1. Write a brief description of the test as the descriptive label in the testing function or as a comment

2. If there are any global state variables to use in the test, set them to specific values.
3. If you expect user input from the keyboard, call `check.set_input`.
4. If you expect your function to print anything to the screen, call `check.set_screen` or `check.set_print_exact`.
5. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`).
6. If the effects for the function include mutating global state variables or parameters, call the appropriate `check` function after the return value test, once for each mutated value to be checked.

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

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

Notice below how the helper function does not need examples.

6.4.1 Module 4: Mutation

```
##
## ***************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 4
## ***************************************************
##

import check


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

    Effects: Mutates lst

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

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


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

    Effects: Mutates lst

    add_one_to_evens: (listof Int) -> None

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

       if L = [3,5,-18,1,0],
       add_one_to_evens(L) => None,
       and L = [3,5,-17,1,1]
    '''
    add_ones_to_evens_by_position(lst, 0)


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

L = [3,5,-18,1,0]
check.expect("Empty list", add_one_to_evens(L), None) #check return
check.expect("Empty list (checking L)", L, [3,5,-17,1,1]) # check mutation


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

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

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

6.4.2 Module 4: Mutation

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 4
## ************************************************
##

import check


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

    add_evens: (listof Int) -> Nat

    Examples:
      add_evens([]) => 0,
      add_evens([3,5,-18,1,0]) => -18,
    '''
    add = 0
    if lst == []:
        return 0
    elif lst[0] % 2 == 0:
        add = lst[0]
    return add + add_evens(lst[1:])


##Examples:
L = []
check.expect("Empty list", add_evens(L), 0) #check return
check.expect("Empty list (checking L)", L, []) # check non-mutation
L = [3,5,-18,1,0]
check.expect("Empty list", add_evens(L), -18) #check return
check.expect("Empty list (checking L)", L, [3,5,-18,1,0]) # check non-mutation


##Tests:
L = [2]
check.expect("One even number", add_evens(L), 2) #check return
check.expect("One even number (checking L)", L, [2]) #check non-mutation
L = [7]
check.expect("One odd number", add_evens(L), 0) #check return
check.expect("One odd number (checking L)", L, [7]) #check non-mutation
L = [1,4,5,2,4,6,7,12]
check.expect("General case", add_evens(L), 28) #check return
check.expect("General case (checking L)", L, [1,4,5,2,4,6,7,12]) #check non-mutation
```

7 Module 5

7.0.1 Module 5: Accumulative Recursion

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 5 Accumulative Recursion
## ************************************************
##

import check

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

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

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

    factorial: Nat -> Nat
    Requires: r >= 0.0

    Examples:
      factorial(0) => 1
      factorial(1) => 1
      factorial(5) => 120
    '''
    return remember_fact(1, n)

##Examples:

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

##Tests:

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

7.0.2 Module 5: Generative Recursion

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 5 Generative Recursion
## ************************************************
##

import check

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

    is_palindrome: Str -> Bool

    Examples:
        is_palindrome("") => True
        is_palindrome("a") => True
        is_palindrome("abba") => True
        is_palindrome("abca") => False
    '''
    if len(s) < 2:
        return True
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])

##Examples:

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

##Tests:

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

7.0.3 Module 5: Local Helper Function

Notice how the helper function awkwardly doesn't include the parameter `n` but must reference it to make sense of what the function does. The local helper function can access parameters that are part of the containing function.

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

import check


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

    fib: Nat -> Nat

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

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

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

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


##Examples:

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

```
check.expect("Example 3: f10", fib(10), 55)

##Tests:

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

8 Module 9

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

8.1 Python Classes (Module 9)

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

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

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

The description for `__init__` should include the statement

```
'''
Constructor: Create a Time object by calling Time(h, m, s)
'''
```

followed by the contract and requirements:

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

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

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

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

8.1.1 Module 9: Dictionaries

```
##
## ************************************************
##   Rita  Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 9 Dictionaries
## ************************************************
##

import check


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

    character_count: Str -> (dictof Str Nat)

    Examples:
        character_count('') => {}
        character_count('a') => {'a':1}
        character_count('banana') => {'a':3, 'b':1, 'n':2}
    '''
    characters = {}
    for char in sentence:
        if char in characters:
            characters[char] = characters[char] + 1
        else:
            characters[char] = 1
    return characters


##Examples:

check.expect("Example 1: Empty", character_count(''), {})
check.expect("Example 2: Singleton", character_count('a'), {'a':1})
```

```
##Note that the order of the dictionary below does not matter
check.expect("Example 3: Typical", character_count('banana'), {'n':2, 'a':3, 'b':1})
check.expect("Example 3: Typical", character_count('banana'), {'a':3, 'b':1, 'n':2})

##Tests:

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

8.1.2 Module 9: Classes

```
##
## ************************************************
##   Rita Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 9: Classes
## ************************************************
##

import check

##Useful constants
seconds_per_minute = 60
minutes_per_hour = 60
seconds_per_hour = seconds_per_minute * minutes_per_hour

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

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

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

        __init__: Time Nat Nat Nat -> None
        Requires: 0 <= h < 24, and 0 <= m, s < 60
        '''
        self.hour = h
        self.minute = m
        self.second = s

    def __repr__(self):
        '''
        Returns a string representation of self
        (Implicitly called by print(t), where t is of time Time)

        __repr__: Time -> Str
        '''
        #The 0:0=2d means display 2 digits.
        return "{0:0=2d}:{1:0=2d}:{2:0=2d}".format(
                      self.hour, self.minute, self.hour)

    def __eq__(self, other):
```

```
        '''
        Returns True if self and other are considered equal, and False otherwise
        (Implicitly called when for t1 == t2 or t1 != t2, where
         t1 is a Time value, and t2 is of type Any)

        __eq__: Time Any -> Bool
        '''
        return isinstance(other, Time) and \
               self.hour == other.hour and \
               self.minute == other.minute and \
               self.second == other.second

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

       time_to_seconds: Time -> Nat

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

       if just_before_midnight is Time(23,59,59),
       then just_before_midnght.time_to_seconds(t) => 86399
       '''
       return (seconds_per_hour * self.hour) + \
              seconds_per_minute * self.minute + self.second


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

  earlier: Time Time -> Bool

  Examples:
     just_before_midnight = Time(23, 59, 59)
     noon = Time(12, 0, 0)
     earlier(noon, just_before_midnight) =>True
     earlier(just_before_midnight, noon) => False
  '''
  return time1.time_to_seconds() < time2.time_to_seconds()

##A sample test for __init__
midnight = Time(0, 0, 0)
check.expect("Example __init__ hour", midnight.hour, 0)
check.expect("Example __init __minute", midnight.minute, 0)
check.expect("Example __init__ second", midnight.second, 0)

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

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

##A sample test for __eq__
midnight = Time(0, 0, 0)
midnight2 = Time(0, 0, 0)
notmidnight = Time(0, 0, 1)
check.expect("Example __eq__", midnight == midnight2, True)
check.expect("Example __eq__", midnight == notmidnight, False)

##Examples for time_to_seconds:

##useful values 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)

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


##Tests for time_to_seconds:

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

##Examples for earlier:

check.expect("before", earlier(noon, just_before_midnight), True)
check.expect("after", earlier(just_before_midnight, noon), False)

##Tests for earlier:
check.expect("before", earlier( midnight, eight_thirty) ,True)
check.expect("after", earlier( eight_thirty, midnight) ,False)
check.expect("just before", earlier( eight_thirty, eight_thirty_and_one),
             True)
check.expect("just after", earlier( eight_thirty_and_one, eight_thirty),
             False)
check.expect("same time", earlier( eight_thirty_and_one, eight_thirty_and_one),
             False)
```

9 Module 10

In module 10, we introduce file input and output. This once again comes with updates to effects and testing sections:

9.1 Purpose

Actions Function perform may include producing a value, mutations, the use of `input` and `print`, as well as any file operations.

9.2 Contracts

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

9.3 Effects

Our effects now can include:

- Printing to screen
- Taking input from user
- Mutating a parameter (eg. a list or dictionary or class object)
- Reading a file
- Writing to a file

9.4 Examples

We also need to make additions to examples to account for file input and output. Do your best to describe what is printed to files and what is required from files.

```
'''
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
'''
```

9.5 Testing

At least we can now state our full testing module

1. Write a brief description of the test as the descriptive label in the testing function or as a comment (including a description of file input when appropriate).
2. If there are any global state variables to use in the test, set them to specific values.
3. If you expect user input from the keyboard, call `check.set_input`.
4. If you expect your function to print anything to the screen, call `check.set_screen` or `check.set_print_exact`.
5. If your function writes to any files, call `check.set_file_exact`.
6. Call either `check` function (`expect` or `within`) with your function and the expected value (which may be `None`).
7. If the effects for the function include mutating global state variables or parameters, call the appropriate `check` function after the return value test, once for each mutated value to be checked.

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 comment with a description of what is contained in the files read in that test.
- Step 5: If your function writes to a file, you will need to use the command `check.set_file_exact` before running the test. The function consumes two strings: the first is the name of the file that will be 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 in addition to comparing the expected and actual returned values. If the files are exactly the same, the test will print nothing; if they differ in any way,

the test will print which lines don't match, and will print the first pair of differing lines for you to compare.

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

9.6 Module 10: File Input and Output

```
##
## ************************************************
##   Rita Sanchez (12345678)
##   CS 116 Fall 2017
##   Module 10 File Input and Output
## ************************************************
##


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

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

    file_filter: Str Int -> None
    Requires:
       0 <= minimum <= 100
       fname exists

    Examples:
       If "empty.txt" is empty, then file_filter("empty.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)
    '''
    infile = open(fname, "r")
    lst = infile.readlines()
    infile.close()
    outfile = open("summary.txt", "w")
    for line in lst:
        if int(line.strip()) > minimum:
            outfile.write(line)
    outfile.close()

##Test 1: empty file (example 1)
```

```
check.set_file_exact("summary.txt", "empty.txt")
check.expect("t1", file_filter("empty.txt", 40), None)
##Test 2: small file (example 2)
#eg2-summary contains 75 and 90 once per line.
check.set_file_exact("summary.txt", "eg2-summary.txt")
check.expect("t2", file_filter("ex2.txt", 50), None)
##Test 3: file contains one value, it is > minimum
check.set_file_exact("summary.txt", "one-value.txt")
check.expect("t3", file_filter("one-value.txt", 20), None)
##Test 4: file contains one value, it is < minimum
check.set_file_exact("summary.txt", "empty.txt")
check.expect("t4", file_filter("one-value.txt", 80), None)
##Test 5: file contains one value, it is minimum
check.set_file_exact("summary.txt", "empty.txt")
check.expect("t5", file_filter("one-value.txt", 50), None)
##Test 6: file contains 1-30 on separate lines
check.set_file_exact("summary.txt", "sixteen-thirty.txt")
check.expect("Q3T4", file_filter("thirty.txt", 15), None)
```