Exam 2 Study guide

(Note: this is mostly Dr. Johnson's study guide for the majors' sections. When I read it, I realized - hey, this is pretty darned applicable for our section, too. I made a few changes, but not many. Giving credit where credit is due.)

This study guide covers *the material you are responsible for on the second exam*. This guide is expressed in terms of skills you must be able to demonstrate on the exam. You are going to be expected to evaluate, write, and examine Python code to demonstrate the skills covered in this study guide. Additionally, there will be "normal" questions (multiple choice, short answer, etc.) where you will demonstrate your understanding of the material.

Exam 2 is similar to Exam 1, except that it is worth more points. Exam 2 will be administered on Blackboard, and will be an open book/open notes exam. You will have 90 minutes to complete the exam. I will try to put a "practice exam" on Blackboard ahead of class so that, if you have never had the experience of taking a test on Blackboard, you can get some practice.

Computer science is by nature ***cumulative***. Thus, concepts from the first exam will be present on this exam, though they will not be the focus. You're not allowed to forget how a "for each" loop works

The order, length or depth of a section in this study guide is ***no indication*** *of the relative importance of that topic or its likelihood to appear on the exam.* i.e. Don't ignore the short sections!

While Loops

- *Define* sentinel values, particularly in terms of constants - sentinel is a value that stops the list; often a CONSTANT
- *Define* boolean flags and explain their use in code -"while something is true" or "while something is false" - the thing you're waiting to be true or false is a boolean flag
- Given a loop, *identify* any sentinel values or boolean flags in use

Constants

- *Define* magic numbers and *explain* why they are undesirable, in terms of:
    - Code readability (how easy it is to read)
    - Code maintainability (how easy it is to change)
    - Magic number is a literal that's an int or float; used in program with no explanation about what it is
- *Compare and contrast* literals and magic numbers, taking into consideration:
    - When is a magic number a literal?

    - When is a literal a magic number?
    - Magic number is a literal; it’s a number. Literals can also be strings - a string literal
- Given a code snippet, *identify* which literals are magic numbers, and which are not
- *Define* constants. *Explain* their relationship with magic numbers
    - Values that do not change within a program run. Constants should be used to replace magic numbers and make code easier to follow
- *Define* the rules for naming constants
    - ALL CAPS
    - DESCRIPTIVE
- *Compare and contrast* constants and variables
    - In Python, constants are not natively supported
    - We use coding standards to distinguish
    - Variables change; constants don’t

Strings

- Define concatenation
    - Combining two strings: use a + sign ‘UM’+’BC” -> ‘UMBC’
- Define slicing and substrings and explain their relationship
    - Slicing - taking some or all characters in a string using [a:b]
    - If first number is blank[:b] start at the beginning
    - If second number is blank [a:] go to end
    - Use slicing to get a substring
- Define escape sequences and enumerate the escape sequences for:
    - A tab character \t ‘\t’ ‘\\t’ print out \t, not a tab
    - A newline character \n ‘\n’ ‘\\n’ prints out \n
    - A single or double quote character \’ \”
    - A backslash \\
- Define whitespace
    - Blanks, newlines, tabs
- Implement code that:
    - Accesses individual characters in a string - s[6]
    - Gets a copy of the string that is all lower case s.lower()
    - Gets a copy of the string that is all upper case s.upper()
    - Gets a substring that
        - Starts and ends in the middle - a:b
        - Starts at the beginning and ends somewhere in the middle [:b]
        - Starts somewhere in the middle and ends at the end [a:]
    - Remove whitespace at the front and end of a string using strip(), lstrip() and rstrip()
    - Split a string into a list of substrings, using `split()`
    - Join a list of strings into a single string, using `join()`

        - `A = '+'.join(b) - list into string A = '1+2+3'`
        - `A = ''.join(b) b=[1,2,3] - A = '123'`

Functions

- Describe functions as the fourth type of program control
    - Sequential - one statement at a time in order
    - Conditional - if something is true or false
    - Iterative - loops; do something multiple times
    - Functions - jump from where you are to some other part of the program
- Define code duplication and explain how it causes:
    - More bugs
    - Harder to maintain code
    - ***Don't write the same code multiple times***

- Define the parts of a function and identify them in code:
    - Name
    - Parameters
    - Body
    - Definition
    - Arguments
    - Call
    - Return statement
- Explain what a return statement does in terms of a function and its caller - passes one value (maybe a list) back to caller from function
- Identify where functions can be legally called in code - anywhere except the left-hand-side of an assignment - cannot say get_file(filename) = x + 3 - 4
- Identify problems that occur when a called function does not return a value when the calling code expects it to. i.e. Consider **`print(sum(x, y))`** but sum does not have a return statement - no return statement - return None - can't use that in a lot of code
- Explain what receiving an error containing NoneType indicates - function returned None
- Define `None` in terms of when it will appear while you debug your code
- Define scope and local variables. Describe what errors may occur if a function tries to access another function's local variables. Local variables - defined in function. Scope - where in the program you can use a variable. Main program variables are global in scope - can be used anywhere.

Program Design

- Define modularity and explain the benefits of a modular program
- Define helper functions
- Given a large problem, outline a program design that breaks down the problem into subproblems

- Define, compare and contrast top-down implementation and bottom-up implementation. (Hint: do so in terms of a program design outline)
- Advocate for testing as you go
- Define readability and explain its benefits
- Improve the readability of supplied code by applying the methods listed above
- Advocate for testing as you go
- Provide examples of good places to comment code
- Identify excessive comments in supplied code
- Advocate for testing as you go

2-dimensional Lists

- Define a 2 dimensional list
- Explain how to and/or implement:
  - Instantiate an empty 2d list l1 = []
  - Instantiate Height x Width 2d list with a particular value (or set of values)
    - L2 = [[1,2,3],[4,5,6]]
  - Access/Iterate over a row in a 2d list - rows are FIRST subscript; can do a row at one time
  - Access/Iterate over a column in a 2d list - you MUST use a loop, access one row/column at a time.
  - Access a single value in a 2d list
    - Which [] is the row number?
    - Which is the column number?
- Given a 2d list, access a particular value or set of values from the list - second row, third column in the list - l1[1][2]
- Explain why a 1d list of strings can be considered a 2d list - strings are “sort of” lists

Mutability

- Define mutable, and give examples of type(s) that are mutable - lists; dictionaries - can be changed after initial declaration
- Define immutable, and give examples of type(s) that are mutable - cannot be changed after declaration - int, float, string, boolean
- Compare and contrast assignment of mutable and immutable variables - a is a list; b = a; b and a point to the same location in memory, so changing b changes a. Doesn’t work that way for immutable variables.
- Define, compare and contrast deep and shallow copy
  - Remember that “deep copy” of a 2-d list is more complicated than “deep copy” of a 1-d list.
- Implement a deep copy in the following ways: a = [1,2,3]
  - With a loop
    - For i in range(len(a)):

            - b[i] = a[i]
    - With list() b = list(a)
    - With sllicing b = a[:]
- Explain the impact of mutable parameters to a function in terms of shallow copy and scope argument to parameter mapping works the same as assignment
- B = a   assignment; b & a point to same place in memory
- In main program : test(b)
- In function: def test(a):   a&b are lists; they point to same location in memory

Dictionaries

- *Define* a dictionary in terms of association, ordering, keys and values
- *Explain* the constraint applied to keys with respect to mutability and uniqueness
- *Explain* why there are less constraints on values
- *Read, write and delete* entries from a dictionary
- *Define* and *utilize* the following dictionary methods:
    - `keys()`
    - `values()`
    - `get()`