Fall 2026 Midterm 2 Concepts

These are concepts for the Fall 2026 Midterm 2 Review Guide.

The Midterm 2 exam covers all material in the course up to and including Lec 18 (Recursion I) and Lab 11 (Recursion).

Because the material tested in Midterm 2 builds upon all the material tested in Midterm 1, Midterm 2 is **cumulative** and necessarily covers all the concepts from Midterm 1.

> Here is the list of concepts that were covered in Midterm 1.

However, Midterm 2 will primarily focus on the newer concepts from the course that were *not* emphasized on Midterm 1. Below is a list of those concepts.

Lists

- A **list value** is a **mutable sequence** of values.
- A list is printed as comma-separated **values** delimited by square brackets, for example `[8, 2, 5]`, `['abc', 'd', 'ef', '']`, `[17, 'four', True, [None, 3.141]]`. Observe from these examples:
  - Strings in a list **must** be quoted.
  - The values in a list may have different types.
  - Lists can be nested within other lists.
- A **new list value** can be created by comma-separated **expressions** delimited by square brackets. For example, `[2*3, 'ab' + 'c', 5>6]` creates the new list value `[6, 'abc', False]`.
  - Other ways to create new list values are (1) **concatenating** two lists with + (see below) and (2) **slicing** (see below).
- Standard sequence operations and functions work on lists:
  - `len` returns the number of slots (or values) in the list
  - + concatenates two lists, returning a **new** list that has the elements of both.
  - Subscripting via **`L[`*index*`]`** returns the value in the slot of list `L` at that index.
    - Negative indices from **`L[-1]`** to **`L[-len(L)]`** return the elements of `L` from the end of L to the beginning.
    - Any attempt to use an index outside the range `-len(L)` to `(len(L)-1)` generates an `IndexError`.
  - Slicing `L` via **`L[`*start*`:`*stop*`:`*step*`]`** returns a new list consisting of the elements from **`L[`*start*`]`** up to (but not including) **`L[`*stop*`]`** changing the index by *step*.
    - If *step* is omitted, it defaults to 1.
    - If *step* is positive, there is **no error** if *stop* `>= len(L). Instead,` invalid indices >= len(L) are simply ignored. Similarly, if *step* is negative, there is **no error** if *stop* `< -len(L).`
    - **`L[:`*n*`]`** is a shorthand for **`L[0:`*n*`]`**, which makes a list containing the first *n* values of `L.`
    - **`L[`*n*`:]`** is a shorthand for **`L[`*n*`:len(L)]`**, which makes a list containing all but the first *n* values of `L.`
    - **`L[:]`** is a shorthand for **`L[0:len(L)]`**, which makes a copy of the list `L.`
    - **`L[::-1]`** is a shorthand for **`L[-1:-len(L):-1]`**, which makes a new list that has the values of `L` in reverse order.
  - *val* `in` *list* returns `True` if *val* is one of the values in *list* and `False` otherwise.
    - *val* `not in` *list* does the opposite.
- The `list` function takes any sequences and converts it to a list of the values in that sequence.

- Lists are mutable because the values in their slots can change over time. Indeed, a list can be viewed as an indexed collection of mutable variables.
    - The value in the slot of L at index i can be changed to *newValue* via the assignment **`L[i] = newValue`**.
    - **`L.pop(i)` removes** the slot in L at index i and **returns** the value that was formerly contained in `L[i]`.
        - **`L.pop()` i**s a synonym for **`L.pop(-1)`** and **`L.pop(len(L)-1)`**
    - **`L.insert(i, val)`** adds a new slot in L at index i containing value ***val***. All slots in L formerly with *index* >= i are renumbered to have index *index+1.*
        - If i >= `len(L)`, there is no IndexError. Instead, a new slot is added at the **end** of the list. Similarly, If i < `-len(L)`, a new slot is added at the beginning of the list.
        - **`L.insert(i, val)` returns** `None`, which is often a gotcha! It does ***not*** returned the modified list!
    - **`L.append(val)`** adds a new slot in L at the end of the list (at index `len(L)`) containing ***val***.
        - **`L.append(val)`** is a synonym for **`L.insert(len(L), val)`**
        - **`L.append(val)` returns** `None`, which is often a gotcha! It does ***not*** returned the modified list!

Ranges

- range generates a sequence of integers, can be used in three ways:
    - `range(end)` generates a sequence from 0 to start-1,eg `range(5) produces [0,1,2,3,4]`
    - `range(start:end)` generates a sequence from 0 to end-1, eg `range(4:8) produces [4,5,6,7]`
    - `range(start:end:step)` generates a sequence from start to step-1, incrementing by step, eg `range(3,25,5)` produces `[3, 8, 13, 20]` (note that step can be negative, eg `range(10, 1, -2)` produces `[10, 8, 6, 4, 2]`

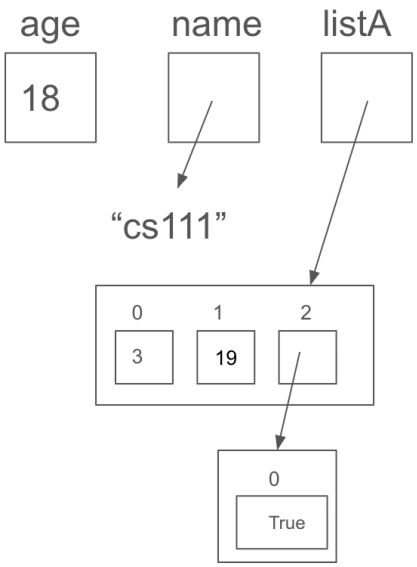
Memory Diagrams

- Memory diagrams show the state of a program involving variables and values, including mutable values like lists and dictionaries.
- They are especially important for understanding:
    - **assignment**: changing the value stored in a variable, list slot, or dictionary slot
    - **adding or removing slots** in a list or dictionary
    - **aliasing**: the same mutable value can be accessed by multiple paths.
    - **==** vs. **is** for testing value equality.
- You should be able to do the following:
    - Given Python code involving variables, lists, and/or dictionaries, draw a memory diagram that represents the state of memory after the code is executed.
    - Given a memory diagram, write Python code that creates the state indicated by the memory diagram.
    - Given memory diagrams *MD1* and *MD2*, write Python code that transforms MD1 to MD2, possible satisfying other constraints (e.g., not creating any new lists/dictionaries, using only particular variables in your code).
- Here are the guidelines for drawing memory diagrams:
    - Each variable is represented by a box, labeled with the variable name
        - `numbers, booleans, None` can go directly in the slot boxes or variables
        - `strings` are drawn outside the slot box, with an arrow pointing to the string
        - `lists` are drawn as a big box, with a slot box (labeled with index) for each element in the list

- arrows in a memory diagram can point to 1) a `string` or 2) to anywhere on the border of a "big" box (i.e., a box for lists/tuples/dictionaries that contains smaller slot boxes). Arrows can *never* point to a small box (i.e., a variable box or slot box in a list/tuple/dictionary).
-

Below are examples of three variables: `age, name`, and `listA`.

```
age = 18
name = 'cs111'
listA = [3, 19, [True]]
```



-

List Accumulation Patterns

- **Mapping:** given a sequence *seq*, a new list consisting of a function `f` applied to each element of L can be created via the following pattern:

```
newList = []
for val in seq:
    newList.append(f(val))
# Do something with newList here
```

- **Filter:** given a list L, a new list consisting of only those elements satisfying a predicate pred can be created via the following pattern:

```
newList = []
for val in seq:
    if pred(val):
        newList.append(val)
# Do something with newList here
```

- **Combined Mapping and Filtering:** filtering with `pred` and mapping the results with `f` can be combined in a single loop:

```
newList = []
```

```
for val in seq:
    if pred(val):
        newList.append(f(val))
# Do something with newList here
```

Tuples

- A **tuple value** is an **immutable sequence** of values.
- Tuples are printed as comma-separated **values** delimited by parenthesis, for example: `(8, 2, 5)`, `('abc', 'd', 'ef', '')`, `(17, 'four', True, (None, 3.141), [23, 'four', False])`, `(42,)`. Observe from these examples:
  - Strings in a tuple **must** be quoted.
  - The values in a tuple may have different types.
  - Tuples can contain not other tuples, but can also contain mutable values like lists (and dictionaries).
  - A tuple containing only one value is written as `(`*value*`,)`, because otherwise it could not be distinguished from the parenthesized value `(`*value*`,)`.
- A **new tuple value** can be created by comma-separated **expressions** delimited by parentheses. For example, `(2*3, 'ab' + 'c', 5>6)` creates the new tuple value `(6, 'abc', False)`.
  - A new tuple with only one value is created with the syntax `(`*expression*`,)`, because otherwise it could not be distinguished from the parenthesized expression `(`*expression*`,)`.
  - In some contexts, the parentheses for creating new tuples are optional. For example, `tup = (1+2, 3*4)` can be written `tup = 1+2, 3*4` and `return (1+2, 3*4)` can be written `return 1+2, 3*4`. However, when tuples are passed as arguments to functions or used as elements in a list or another tuple, the parentheses are **always** required. So it's safest to always use explicit parentheses around tuples!
  - Other ways to create new list values are (1) **concatenating** two tuples with + and (2) **slicing** the tuple (using the same notation as list slicing).
- Because tuples are **immutable**, the following list operations **cause errors** if they are attempted on tuples:
  - slot assignment: e.g. *tuple*`[`*index*`] = `*expression*
  - `.pop`
  - `.append`
  - `.insert`
- Tuples support the following sequence operations:
  - Tuple subscripting/indexing: e.g. *tuple*`[`*index*`]`, which evaluates to the value in slot *index* of *tuple*.
  - `len`
  - Concatenation of two tuples with `+`
  - Slicing of tuples
  - *val* **`in`** *tuple* and *val* **`not in`** *tuple*

Tuple Assignment

- If *seqExp* is an expression denoting a *n*-element sequence and var_0, var_1, …, var_{n-1} are *n* distinct variables, then the **tuple assignment**

  var_0, var_1, …, var_{n-1} `=` *seqExp*

  is equivalent to the following sequence of statements:

  seqVar `=` *seqExp*
  var_0 `=` *seqVar*`[0]`

```
var1 = seqVar[1]
...
varn-1 = seqVar[n-1]
```

  - *var*$_0$, *var*$_1$, …, *var*$_{n-1}$ can be wrapped in parens, but the parens are optional.
- Tuple assignment makes it easy to name the elements in a list, tuple, string, or range if the length of that sequence is known, without first having to name the whole sequence. For example:
  - `x, y, z = (10, 3, 24)`
  - `(c1, c2, c3, c4) = 'pear' # Uses optional parens`
  - `name, age, isCSMajor = ('Georgia Dome', 19, True)`
- Tuple assignment makes it easy to simultaneously change variable values, as in `x,y = y,x`
- Tuple assignment can be used in for loops and list comprehensions, as in

```
for x,y in [(7, 3), (2, 4), (1, 8)]
    print(x + y)
```

```
[x + y for x,y in [(7, 3), (2, 4), (1, 8)]]
```

Sorting

- Given any sequence value *seqVal*, the `sorted(`*seqVal*`)` returns a **new list** whose elements are sorted in **ascending** order:
  - Numbers are sorted low to high.
  - Characters are sorted by their ASCII values (dictionary order for letters, but more complex for digits, punctuation, etc.)
  - Strings are sorted in dictionary order (using the ASCII values for characters):
  - Tuples
    - are first sorted by the values at their lowest index
    - if there is a tie at index *i*, the value at index *i*+1 is used as a tie-breaker (recursively, if necessary)
    - If *tuple*$_1$ is a prefix of *tuple*$_2$, then *tuple*$_1$ comes before *tuple*$_2$ in ascending order.
- Given a list `L`, `L.sort()` mutates the list `L` so that the elements of `L` are in ascending order.
  - `L.sort()` returns the `None` value.
- Both `sorted` and `.sort` accept an optional keyword argument `reverse=`*booleanExpression*. If *booleanExpression* evaluates to a **truthy** value, the list values are sorted in **descending** order; otherwise the values are sorted in **ascending** order.
- Both `sorted` and `.sort` accept an optional keyword argument `key=`*functionExpression.* Suppose that the values being sorted are *listVals* and the value of *functionExpression* is the function *keyFcn*. Then the order of the values in the sorted list with `key=`*functionExpression* is determined by the ordering in this list comprehension expression:

```
[val for decoration, val in sorted([(keyFcn(val), val) for val in listVals])]
```

  This means that the values will be sorted by *keyFcn*`(val)` rather than the natural ordering of `val`.

Nested Loops

- Any kinds of loops can be arbitrarily nested.
- All iterations of an inner loop must complete for the next iteration of an outer loop can begin, so inner loops proceed more quickly than outer loops.

- Nested loops are particularly natural when processing data structure in which some combination of lists, dictionaries, and strings are themselves nested.
- You should be able to read and write programs involving nested loops.

Files

Reading Content from a file:

- `.readlines()`: Reads all lines and returns them as a list of strings
  - Each string in the list ends in a newline character; use `.strip()` to remove this newline.
- The preferred way to read content is to use a `for` loop over each line in a file opened for reading, as in this template:

```
for line in inputFile:
    … statements that use line …
```

Writing Content:

- write(): Writes a specific string to the file
- **Warning:** Unlike print, .write does **not** add a default newline to the end of the argument that is written. It is necessary to include an **explicit** newline character after the argument if that's what you want.

We use this syntax:

```
with open(filename, mode) as fileObj:
  … statements that use fileObj …
```

where mode is one of these letters:

- `'r'` for read (default mode)
- `'w'` to write to a file (overwrites any existing file with filename)
- `'a'` appends to existing contents of filename

A key benefit of the `with open … as:` syntax is that the file is automatically closed when the program is deindented to the same indentation level as the **`with`** keyword.

Examples of **`with`** `open … as:`

yummy.txt

```
ice cream
pizza
boba
```

`with open('yummy.txt','r') as fileObj:` `  listOfStrings = fileObj.readlines()`	`listOfStrings` contains: `["ice cream\n", "pizza\n", "boba\n"]`

Exception Handling

- Normally, an error/exception in a program causes the program to terminate. But errors/exceptions can be handled using the **`try`**/**`except`** statement, which allows programs to continue executing even when errors/exceptions occur:

  `try`:
  try-statements
  `except` *exceptionType*$_1$:
  except-statements$_1$
  …
  `except` *exceptionType*$_n$:
  except-statements$_n$

  The above **`try`**/**`except`** statement executes as follows:
  - All statements in *try-statements* are executed.
    - If no errors/exceptions occur during the execution of *try-statements* none of the *except-statements* are executed.
    - If an error/exception occurs during the execution of *try-statements,* then *except-statements*$_i$ is executed for the first *exceptionType*$_i$ that matches the type of the error/exception.
  - Any of the *except-statements*$_i$ might contain errors that could terminate the program or be handled by other **`try`**/**`except`** statements wrapped around this one.
- Common error/exception types in this course are `TypeError`, `ValueError`, `NameError`, `IndexError`, `KeyError`, and `FileNotFound` error.
  - `SyntaxError` cannot be handled because it is a more fundamental kind of error that prevents the program from being executed.

Dictionaries

- Dictionaries are mutable collections of key/value pairs.
  - The keys of a dictionary must be:
    - Immutable values (numbers, strings, booleans, `None`, tuples)
    - Unique within a given dictionary
  - The values in a dictionary can be anything, including another dictionary (they can be nested)
  - Inserting one key:value pair at a time can be done like this:
    - `myDict[key] = value`
  - Dictionaries are **key**-based, so the `in` operator only looks at keys, and a for loop over a dictionary looks at the keys.
  - `len` works with a dictionary, it returns how many key:value pairs are in the dictionary
  - Dictionaries are represented by { },
    eg `myDict = {'keyA': 'valueA', 'keyB': 'valueB', 'keyC': 'valueC'}`
- Use subscripting notation, [...], to get and set value for key.
- `.keys(), .values(), .items(), .get()` methods are helpful
- common use is to keep the frequency count of items

Recursion (The midterm will only cover *non-fruitful* recursion.)

- A function is recursive if and only if it calls itself.
  - **base case(s)** are case(s) in which the problem is so small that it can be solved directly.

    - **recursive case(s)** (or **general case(s)**) case(s) in which a large problem is solved by calling the function one or more times on smaller inputs and gluing together the results and/or side effects (including printed output) of the recursive calls to correctly solve the problem for the larger inputs.
        - The notions of *larger* and *smaller* depends on the particular problem.
        - In order for a recursive process to terminate, eventually every path of recursive calls must reach a base case. If this is not true, there will be a so-called **infinite recursion**.
            - In Python, an infinite recursion cannot really go on forever; the recursive process will eventually exhaust memory resources (causing a `RecursionError` or `MemoryError`)
            - In some other languages, some infinite recursions might never terminate, and so act like an infinite loop.
    - Although recursive solutions can always be expressed using loops (potentially in addition to stack-like data structures), in many situations a recursive solution is more concise or elegant than using loops. This is especially true in situations involving tree-shaped data or processes.
- **Function call frame** model for **recursive functions**.
- Analyze + write **recursive functions** to solve problems
- **Invariants** (definition + purpose)