CSC108 Notes Template 8 – Lists

1 Lists: Introduction and Indexing

We have seen some types of objects that are just single items: **`int`**, **`bool`**, **`float`**. We cannot iterate (loop) over them.

We have seen other types of objects that are collections of items: a string is a collection of characters and a Picture is a collection of pixels. We can iterate over these because they are collections of items.

We often want to store collections of items, not just collections of pixels or characters, but other types as well.

Example.

```
measurement1 = 45.27
measurement2 = 45.26
measurement3 = 45.24
...
```

But if we do 100 measurements, we don't want 100 variables! We want one variable that contains all 100 measurements. We can do this in Python with a **`list`**:

```
>>> x = 45
>>> measurement1 = 45.27
>>> measurement2 = 45.26
>>> measurement3 = 45.24
>>> measurements = [45.27, 45.26, 45.24]    # Type list
measurements
>>> [45.27, 45.26, 45.24]
>>> measurements[0]    # element at position 0
45.27
```

```
>>> measurements[1]
45.26
>>> measurements[2]
45.24
>>> measurements[3]
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
IndexError: list index out of range
```

Lists can contain more than just numbers, and even be heterogeneous.

Examples.

```
>>> instructors = ['Andrew', 'Anna', 'Steve']
>>> student = ['Jon Reed', 'Trinity College',
123456789, 3.45]
>>> s = 'hello'
>>> s[0] = 'j'    # strs are immutable
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'str' object does not support item
assignment

# (to be continued...)
```

2 Mutable

see the right block

Lists are mutable (changeable):

```
>>> instructors
['Andrew', 'Anna', 'Steve']
>>> instructors[2] = 'Diane'  # lists are mutable
>>> instructors
['Andrew', 'Anna', 'Diane']
>>> help(id)
Help on built-in function id in module __builtin__:

id(...)
    id(object) -> integer
```

```
Return the identity of an object.  This is guaranteed to be unique among
    simultaneously existing objects.  (Hint: it's the object's memory address.)
>>> id(instructors)
5498240
>>> id(instructors[0])
17220992
>>> id(instructors[1])
17221024
>>> id(instructors[2])
17221792
>>> instructors[2] = 'Karen'
>>> id(instructors[2])
17220864
>>> instructors.append('Paul')
>>> instructors
['Andrew', 'Anna', 'Karen', 'Paul']
```

3 Aliasing

Two variables may refer to the same object. This is called aliasing and we've seen this before:

```
a = 19
b = a
print a, b
print id(a), id(b)
```

Since **ints** are immutable, we can't affect what one variable references by changing the other:

However, lists are mutable, so when two variables refer to the same object and we change that object, they both still refer to the same object:

```
>>> # Aliasing with immutable types
>>> a = 19
>>> b = a
>>> print a, b  # a and b are two names for the same object
19 19
>>> print id(a), id(b)
9285296 9285296
>>> a = 12
>>> print a, b   # changing a does not change b; a refers to a different int
12 19
>>> print id(a), id(b)
9285380 9285296
```

```
>>> # Aliasing with mutable types
>>> a = [1, 2, 3]
>>> b = a
>>> print a, b  # a and b are two names for the same object
[1, 2, 3] [1, 2, 3]
>>> print id(a), id(b)
17233600 17233600
>>> a[0] = 10
>>> print a, b       # a and b still refer to the same list as each other;  the list items change
[10, 2, 3] [10, 2, 3]
>>> print id(a), id(b)
17233600 17233600
>>> print a
[10, 2, 3]
>>> b[1] = 5
>>> print a, b
[10, 5, 3] [10, 5, 3]
```

4 Mutable parameters

Passing a parameter is like an assignment statement. The aliasing that we just saw with ordinary assignment occurs with parameter passing too.

[`mutable_parameters.py`]

```
[evaluate mutable_parameters.py]
[68.4, 45.3, 63.2]
# data and values are two names for the same
list.
# Lots of pictures on the board.
```

When you have a mutable parameter, you can actually change the object it refers to (as opposed to just making a new object). These changes will therefore be changes to the object that the argument refers to, since the argument and the parameter refer to the same object.

4.1 List functions

There are some useful built-in **`list`** functions.

```
>>> # List functions
>>> instructors
['Andrew', 'Anna', 'Karen', 'Paul']
>>> len(instructors)
4
>>> instructors[0]
'Andrew'
>>> instructors[3]
'Paul'
>>> instructors[len(instructors) - 1]   # get the last element of the list
'Paul'
>>> measurements
[45.27, 45.26, 45.24]
>>> min(measurements)
45.24
>>> max(measurements)
45.27
>>> sum(measurements)
135.77
>>> min(23, 45)
23
```

```
>>> min("Hi, class!")
' '
>>> min("abcde")
'a'
>>> sum("abced")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> min("Hi,class!")
'!'
>>> min("Hi,class")
','
>>> min("Hiclass")
'H'
>>> min("iclass")
'a'
```

4.2 List Methods

To get descriptions of the **`list`** methods: **`help(list)`**

With `L = ['a', 'b', 'c', 'd']`, complete the table:

Method	Result	Description
`L.append('e')`	>>>L ['a', 'b', 'c', 'd', 'e']	L.append(object) -- append object to end
`L.insert(2, 'new')`	>>>L ['a', 'b', 'new', 'c', 'd', 'e']	L.insert(index, object) -- insert object before index
`L.insert(len(L), 'z')`	>>>L ['a', 'b', 'new', 'c', 'd', 'e', 'z']	L.insert(index, object) -- insert object before index
`L.sort()`	>>>L ['a', 'b', 'c', 'd', 'e', 'new', 'z']	L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;

del L[1]   /   remove()

| cmp(x, y) -> -1, 0, 1

remove(...) L.remove(value) -- remove first occurrence of value. Raises ValueError if the value is not present.

4.3 `for` loop over a list

```
for item in list:
    body
```

Never actually change the list!

Examples.

```
# Print the items in the list "measurements"
for value in measurements:
   print value
```

Q. What will this print?

```
L = [1, 2, 3]
for item in L:
    item = item * 2
print L
```

A. [1, 2, 3]

Q. How can we loop over a list changing the value of the list items?

A.

```
for item in L:
        item = item * 2
        print item

2
4
6
```

5 range

```
range(...)
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

Examples.

```
range(4)
[0, 1, 2, 3]
range(len(L))
[0, 1, 2]
for i in range(len(L)):
        L[i] = L[i] * 2

print L
[2, 4, 6]
```

```
# by default it increases by one each time

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(4, 10)
[4, 5, 6, 7, 8, 9]
>>> range(4, 10, 2)
[4, 6, 8]
>>> range(4, 100, 5)
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49, 54, 59, 64, 69, 74, 79,
84, 89, 94, 99]
```

6 Slicing and aliasing

Same as for strings, slicing a list creates a new object even if you slice the whole list.

# aliasing

```
>>> subjects = ['CSC', 'Bio', 'French', 'History']
>>> subjects_alias = subjects
>>> print subjects
['CSC', 'Bio', 'French', 'History']
>>> print subjects_alias
['CSC', 'Bio', 'French', 'History']
>>> print id(subjects), id(subjects_alias)
17294600 17294600
>>> subjects[1] = 'Commerce'
>>> print subjects
['CSC', 'Commerce', 'French', 'History']
>>> print subjects_alias
['CSC', 'Commerce', 'French', 'History']
>>> print id(subjects), id(subjects_alias)
17294600 17294600
```

# making a clone

```
>>>subjects = ['CSC', 'Commerce', 'French', 'History']
>>>subjects
    ['CSC', 'Commerce', 'French', 'History']
>>>subjects_clone = subjects[:]
>>>subjects
    ['CSC', 'Commerce', 'French', 'History']
>>>subjects_clone
    ['CSC', 'Commerce', 'French', 'History']
>>>print id(subjects), id(subjects_clone)
    31788288 31788208
>>>subjects[2] = 'Philosophy'
>>>subjects
    ['CSC', 'Commerce', 'Philosophy', 'History']
>>>subjects_clone
    ['CSC', 'Commerce', 'French', 'History']
>>>print id(subjects), id(subjects_clone)
    31788288 31788208
```

7 Practise with lists

[**`list_practise.py`**]

```
def total_length(L):
    '''(list of strs) -> int
    Return the total length of all the strs in L.'''
    total = 0
    for item in L:
        total = total + len(item)
        return total

        #str_lens = []
        #for item in L:
            #str_lens.append(len(item))
        #return sum(str_len)
def square_list(int_list):
    '''(list of ints) -> list of ints
    Return a new list that contains the ints from int_list squared.'''
    squared_list = []
    for item in int_list:
        squared_list.append(item ** 2)
        return squared_list




def to_upper(original_list):
    '''(list of strs) -> NoneType
    Convert all strs in original_list to uppercase.'''
    for i in range(len(original_list)):
        # L still refers to the same list; it's the contents that are changing
         L[i] = original_list[i].upper()
    return L





def get_valid_response(p, valid_list):
    '''(str, list of strs) -> str
    Use p to prompt for and return a string that is from valid_list.'''
    response = raw_input(p)

        while response not in valid_list:
            print 'Valid responses are:', valid_list
            response = raw_input(p)

        return response

    if __name__ == '__main__':

        print get_valid_response("Test today? ", ['yes', 'no'])
        print get_valid_response('What would you like to do?', ['Play Again', 'Save', 'Quit'])
```

8 Nested Lists

We can make lists that contain anything: **ints**, **bools**, **strings**, even other **lists**!

Example.

Let's create a list of student and grade info:

```
grades = [['999888777', 78], ['111222333', 90], ['444555666', 83]]
```

We can refer to elements of the lists within the list as follows:

```
>>> grades = [['111222333', 82],
['999888777', 78], ['444555666', 85]]
>>> len(grades)
3
>>> # grades is a list of [str, int] lists
>>> grades[0]
['111222333', 82]
>>> grades[1]
['999888777', 78]
>>> grades[2]
['444555666', 85]
>>> grades[3]
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
IndexError: list index out of range
>>> grades[0]
['111222333', 82]
>>> grades[0][0]
'111222333'
>>> grades[0][1]
82
>>> for item in grades:
 print item

['111222333', 82]
['999888777', 78]
['444555666', 85]
>>> grades
[['111222333', 82], ['999888777', 78],
['444555666', 85]]
[evaluate nested.py]
>>> average_grade(grades)
81.66666666666667
[evaluate nested.py]
>>> get_student_IDs(grades)
['111222333', '999888777', '444555666']
```

Complete each of the following functions according to their docstrings:

```
def average_grade(grade_list):
```

```
def average_grade(grade_list):
   '''(list of [str, int] lists) -> float
   Return the average grade for all of the
   students in grade_list.'''

   total = 0.0

   # student is a [str, int] list
   for student in grade_list:
      total += student[1]

   return total / len(grade_list)
```

```
def get_student_IDs(grade_list):
```

```
def get_student_IDs(grade_list):
   ''' (list of [str, int] lists) -> list of strs
   Return the student IDs from grade_list.'''

   ids = []

   # item is a [str, int] list
   for item in grade_list:
      ids = ids.append(item[0])
      print ids

   return ids
```

9 Nested Loops

When we have lists inside lists, we can iterate through everything using a loop inside a loop.

```
    measurements = [[33, 34, 30], [29, 31, 34], [28, 27, 30]]
```

```
# Print the average of each inner list
```

```
>>> # nested loops
>>> measurements = [[33, 34, 20],
[29, 31, 34], [28, 27, 30]]
>>> for dataset in measurements:
 print dataset

[33, 34, 20]
[29, 31, 34]
[28, 27, 30]

>>> L = [28, 27, 30]
>>> total = 0
for item in L:
 total += item
>>>
>>> print total / len(L)
28
>>> measurments
Traceback (most recent call last):
  File "<string>", line 1, in
<fragment>
NameError: name 'measurments' is
not defined
>>> measurements
[[33, 34, 20], [29, 31, 34], [28, 27,
30]]
>>> for dataset in measurements:
 total = 0.0
 for datapoint in dataset:
  total += datapoint
 print total / len(dataset)

29.0
31.3333333333
28.3333333333
```

We can do the same thing with builtin function **`sum`**, but it also contains a loop:

Exercise: modify the code to create a **`list`** of the ***averages***.

Exercise: what does the code below print? Trace it in the debugger.

```
for i in range(5):
for j in range(3):
print i, j
```

```
if __name__ == '__main__':
    for i in range(5):
        for j in range(3):
            print i, j
```