11-lists-II

February 20, 2022

0.1 Lists II

In this lecture we look at some functions on lists. For the full documentation of all functions and how they work, see: https://docs.python.org/3/library/stdtypes.html#list and https://docs.python.org/3/tutorial/datastructures.html#more-on-lists.

Python's documentation is your friend. Learn how to use it! You are allowed to consult it during exams.

Disclaimer: most, if not all, techniques below for lists also work for tuples.

0.1.1 Parallel assignment

The elements of tuples can be assigned to variables in parallel:

[1]:
```
T = (4,3,2,1)
a, b, c, d = T
print("a =", a)
print("b =", b)
print("c =", c)
print("d =", d)
```

```
a = 4
b = 3
c = 2
d = 1
```

The above is the same as the less concise:

[2]:
```
T = (4,3,2,1)
a = T[0]
b = T[1]
c = T[2]
d = T[3]
print("a =", a)
print("b =", b)
print("c =", c)
print("d =", d)
```

```
a = 4
b = 3
```

```
c = 2
d = 1
```

You can choose to use either of the above. At this point, there is no preference apart from your own.

The same works with lists, but we advise against that. If you have the wrong number of variables, python will not like it.

[3]:
```
L = [1,2,3]
a, b, c = L
print("a =", a)
print("b =", b)
print("c =", c)
```

```
a = 1
b = 2
c = 3
```

[4]:
```
L = [1,2,3]
a, b = L
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_728660/1149322040.py in <module>
      1 L = [1,2,3]
----> 2 a, b = L

ValueError: too many values to unpack (expected 2)
```

[5]:
```
L = [1,2,3]
a, b, c, d = L
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_728660/730288596.py in <module>
      1 L = [1,2,3]
----> 2 a, b, c, d = L

ValueError: not enough values to unpack (expected 4, got 3)
```

0.1.2 Negative indexing

List indexing admits the use of negative values to indicate positions starting from the end.

[6]:
```
L = [1,2,3,4,5]
last = L[-1]
```

```
second_last = L[-2]
# and so on and so forth
print("last =", last)
print("second_last =", second_last)
```

```
last = 5
second_last = 4
```

0.1.3 Double indexing

Suppose `L` is a list of lists. We can fetch elements from the inner list by first indexing by the list we want, and then by the element we want.

For example, the first element from the second list can be obtained like this: `L[1][0]`.

Note: Also works for tuples.

[7]:
```
L = [ [11,12,13], [21,22,23], [31,32,33] ]
x = L[1][0]
y = L[2][2]
print("x =", x)
print("y =", y)
```

```
x = 21
y = 33
```

You can do this in two steps if you prefer:

[8]:
```
L = [ [11,12,13], [21,22,23], [31,32,33] ]
last_list = L[-1]
x = last_list[0]
# or: x = L[-1][0]
print("x =", x)
```

```
x = 31
```

0.1.4 List comparison

Lists can be compared using the usual boolean operators: `<`, `<=`, `==`, `!=`, `>=`, `>`.

Comparison follows the *lexicographic order* (same way as you would sort words): - compare the first element - if they are the same, compare the second element - if they are the same, compare the third element - ...

Observe that the length of the list is not important, unless all elements are the same.

[9]:
```
L1 = [3,4,6,2]
L2 = [3,2,8,8,8]
print("L1 < L2?", L1 < L2)
print("L1 > L2?", L1 > L2)
```

```
L1 < L2? False
L1 > L2? True
```

[10]:
```
L1 = [3,4,6,2]
L2 = [3,4,6,2,8]
print("L1 < L2?", L1 < L2)
print("L1 > L2?", L1 > L2)
```

```
L1 < L2? True
L1 > L2? False
```

0.1.5 min/max

`min` and `max` are functions on lists and tuples. It is self-evident what they do.

[11]:
```
L = [5,3,4,6,8,9,22,3,563,2,567,3,23,560,148,287,397,517]
mn = min(L)
mx = max(L)
print("min in L is", mn)
print("max in L is", mx)
```

```
min in L is 2
max in L is 567
```

You can call `min` and `max` on a set of values as well:

[12]:
```
mn = min(423,543,134)
mx = max(423,543,134)
print("mn =", mn)
print("mx =", mx)
```

```
mn = 134
mx = 543
```

0.1.6 sum

`sum` is a function that returns the sum of elements of a list or tuple.

[13]:
```
L = [4,5,5,2]
s = sum(L)
print("s =", s)
```

```
s = 16
```

[14]:
```
sum((3,7,9))
```

[14]: 19

0.1.7 sorted

`sorted(L)` returns a list with the elements in `L` in non-decreasing order.

[15]:
```
L = [4,7,5,9,2,5,8]
Ls = sorted(L)
print("Ls =", Ls)
```

```
Ls = [2, 4, 5, 5, 7, 8, 9]
```

Or in non-increasing order if we specify `reverse=True` as one of the arguments.

[16]:
```
L = [4,7,5,9,2,5,8]
Ls = sorted(L, reverse=True)
print("Ls =", Ls)
```

```
Ls = [9, 8, 7, 5, 5, 4, 2]
```

0.1.8 reversed

`reversed(L)` returns an *iterator* to the elements iof `L` in reverse order. Iterator is a construct in python, but you do not need to worry about what it is exactly. Just remember to convert it to a list by using `list(reversed(L))`.

[17]:
```
L = [4,8,1]
Lr = list(reversed(L)) # Remove the list function and see what happens if you␣
 ↪are curious.
print("Lr =", Lr)
```

```
Lr = [1, 8, 4]
```

0.1.9 count

`L.count(x)` counts the number of times the element `x` occurs in the list `L`.

Side note: The dot notation is related to a notion of classes and objects, used in object oriented programming. In this technique, some types are defined in *classes*, and these classes have built-in operators. If `O` is an object of class ("type") `C`, and class `C` defined the operator `op()`, then `op()` on `O` can be invoked as `O.op()`. You do not need to completely understand this at this point in time. Just know that this is a thing.

[18]:
```
L = [4,7,5,9,2,5,8]
fives = L.count(5)
ones = L.count(1)
print("fives =", fives)
print("ones =", ones)
```

```
fives = 2
ones = 0
```

0.1.10 index

`L.index(x)` returns the index of the first occurence of `x` in `L`.

`L.index(x, s)` returns the index of the first occurence of `x` in `L` starting from position `s`.

Note: Observe the dot notation. This means that `index` is a built-in operator in the `list` class.

[19]:
```
L = [4,7,5,9,2,5,8]
index5 = L.index(5)
index5_from3 = L.index(5,3)
print("index5 =", index5)
print("index5_from3 =", index5_from3)
```

```
index5 = 2
index5_from3 = 5
```

0.2 List mutability

List, as opposed to the other types we have seen in this course so far, are *mutable* or *destructable* structures. This means that we can do *in-place* modification of lists (i.e., without assigning it to another variable).

The destructive operators are part of the `list` class, so they are used via the dot notation.

One of the reasons for having destructive operations is to optimize the amount of memory a program takes. Imagine that you have a list with `10 ** 9` elements. Each time you add a new element using `L = L + [x]`, python will create a new list with `(10 ** 9) + 1` elements, by copying all the elements from `L`. This is very inneficient, so python provides a way for you to simply add the element at the end of `L`, modifying it instead of copying all its elements.

In the scope of this course, you will not have to worry about memory efficiency to that level. But this is useful to know about, in case you want to handle really big data.

0.2.1 append

`L.append(x)` will place the element `x` at the end of list `L`, by modifying `L` itself.

This is the destructive version of `L + [x]`.

Side note: Interestingly, and counter-intuitively, `L += [x]` is the same as `L.append(x)`, in the sense that it does not create a copy of the list, and different from `L = L + [x]`.

[20]:
```
L = [3,2,4,5]
print("before L =", L)
L.append(100)
print("after L =", L)
```

```
before L = [3, 2, 4, 5]
after L = [3, 2, 4, 5, 100]
```

0.2.2 extend

`L.extend(L1)` extends the list `L` by list `L1`, by modifying `L` itself. Nothing happens to the list `L1`.

This is the destructive version of `L + L1`.

[21]:
```
L = [3,2,4,5]
L1 = [10,20,30]
print("before L =", L)
print("before L1 =", L1)
L.extend(L1)
print("after L =", L)
print("after L1 =", L1)
```

```
before L = [3, 2, 4, 5]
before L1 = [10, 20, 30]
after L = [3, 2, 4, 5, 10, 20, 30]
after L1 = [10, 20, 30]
```

0.2.3 insert

`L.insert(i,x)` modifies `L` by placing element `x` immediately before element at position `i`.

This is the destructive version of `L[:i] + [x] + L[i:]`.

[22]:
```
L = [0,1,2,3,4]
L.insert(1,42)
print("L =", L)
```

```
L = [0, 42, 1, 2, 3, 4]
```

0.2.4 sort and reverse

`L.sort()` and `L.reverse()` are the destructive versions of `sorted(L)` and `reversed(L)`.

[23]:
```
L = [5,2,8,6,3,0]
L.sort()
print("sorted L =", L)
L.reverse()
print("reversed L =", L)
```

```
sorted L = [0, 2, 3, 5, 6, 8]
reversed L = [8, 6, 5, 3, 2, 0]
```

0.2.5 List assignment

Since lists are built from *classes* (i.e., every individual list is an *object*), it behaves differently from basic types in assignments.

Assignment `L2 = L1` will **not** create a copy of `L1` and store it in `L2`. Instead, it makes the two names `L1` and `L2` refer to the *same* list. That means that, when `L1` is modified, so is `L2`.

This is better illustrated by an example:

[24]:
```
# Behavior for basic types
x = 10
y = x
x = x + 1
print("x =", x)
print("y =", y)
```

```
x = 11
y = 10
```

[25]:
```
# Behavior for objects
L1 = [1,2,3,4]
L2 = L1
L1[0] = 10
print("L1 =", L1)
print("L2 =", L2)
```

```
L1 = [10, 2, 3, 4]
L2 = [10, 2, 3, 4]
```

If you want an actual copy of the list, you can use the `L.copy()` function.

[26]:
```
L1 = [1,2,3,4]
L2 = L1.copy()
L1[0] = 10
print("L1 =", L1)
print("L2 =", L2)
```

```
L1 = [10, 2, 3, 4]
L2 = [1, 2, 3, 4]
```

Side note: If your lists have lists or other complex data-structures inside, *and* you also want those copied, you need to use the `deepcopy()` function.

[27]:
```
# Copy will not create an actual copy of the inner lists
L1 = [[11,12],[21,22],[31,32],[41,42]]
L2 = L1.copy()
L1[0][0] = 100
print("L1 =", L1)
print("L2 =", L2)
```

```
L1 = [[100, 12], [21, 22], [31, 32], [41, 42]]
L2 = [[100, 12], [21, 22], [31, 32], [41, 42]]
```

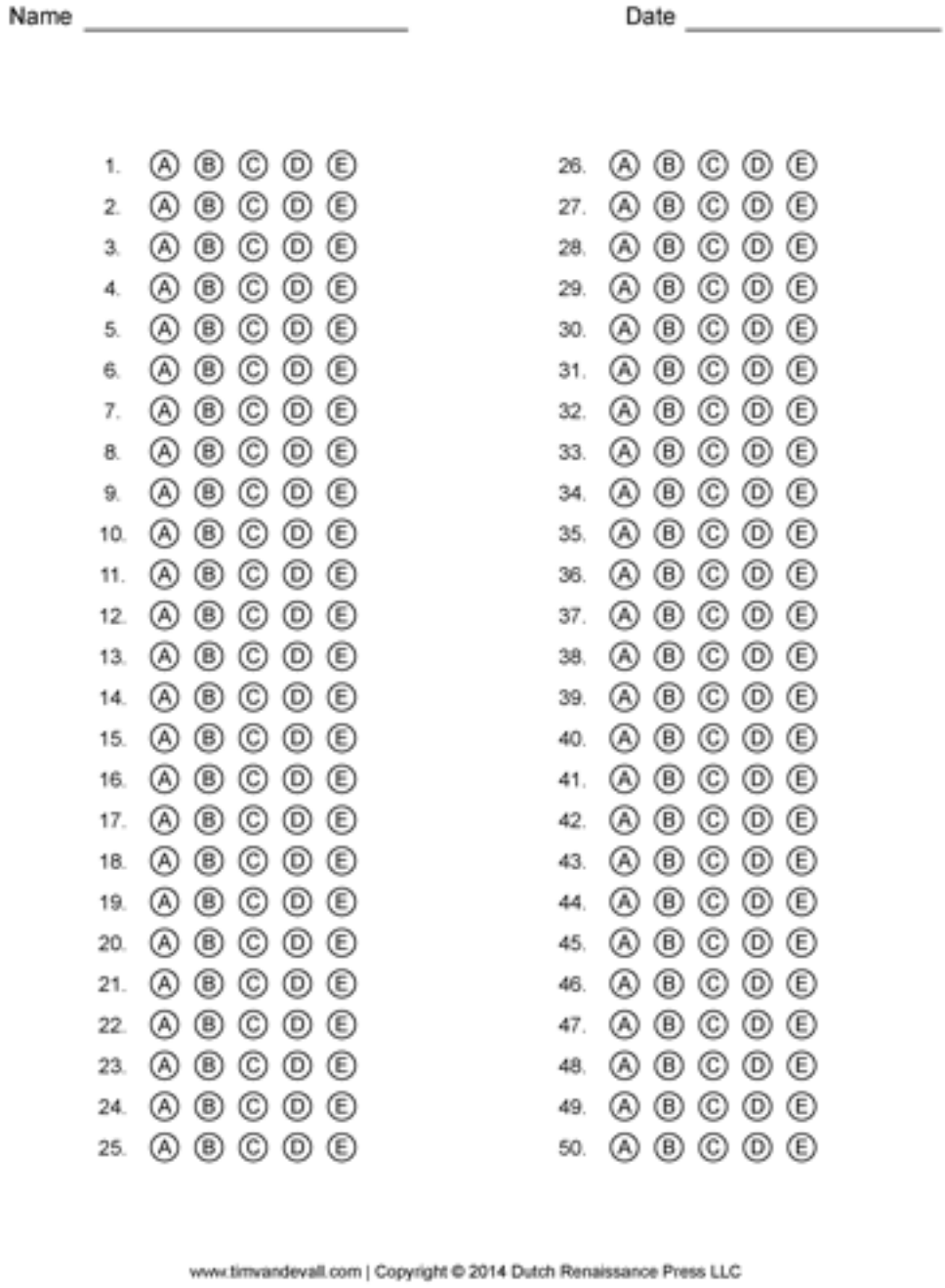
[28]:
```
# But deepcopy will
import copy
```

```
L1 = [[11,12],[21,22],[31,32],[41,42]]
L2 = copy.deepcopy(L1)
L1[0][0] = 100
print("L1 =", L1)
print("L2 =", L2)
```

```
L1 = [[100, 12], [21, 22], [31, 32], [41, 42]]
L2 = [[11, 12], [21, 22], [31, 32], [41, 42]]
```

0.3 Exercise:

Multiple choice exams usually have an answer sheet like this:



After students fill in the answer, an optical reader will scan the paper and assign shades of grey to each bubble of each question. A value of 0 indicates that the bubble is all black (i.e. the student marked that bubble), and a value of 255 indicates that the bubble is all white (i.e. the student did not mark that bubble). Since students fill in the bubbles in various different ways and with pens/pencils of different shades, the convention is that shades less than or equal to 127 are considered marked bubbles, and shades greater than 127 are considered unmarked bubbles.

The result of the scan of one answer sheet is represented as a list of lists. The inner lists have 5

elements, a number between 0 and 255, representing the shades of grey scanned for the bubbles A, B, C, D, and E. Your job is to implement the function `answers(A)` that takes as input the list of lists as described above, and returns a list containing the answers on that answer sheet. The answers are mapped to integers in the following way: 1 for A, 2 for B, 3 for C, 4 for D, and 5 for E. *If no or multiple answers are selected for a question, map it to -1.*

For example, let:

```
A = [ [0,   255, 255, 255, 255],
      [255, 255, 255, 255, 0  ],
      [255, 255, 127, 255, 255] ]
```

then `answers(A)` should return `[1,5,3]`.

[29]:
```
def answers(A):
    return []

A1 = [ [0,   255, 255, 255, 255],
       [255, 255, 255, 255, 0  ],
       [255, 255, 127, 255, 255] ]

A2 = [ [200, 200, 200, 0,   200],
       [200, 1,   200, 200, 1],
       [1,   2,   3,   4,   5],
       [255, 5,   200, 130, 205] ]
```