Lecture 06, 24 August 2023

Nested loops

- find all elements common to `l1` and `l2`
  - for each `x` in `l1`, check if `x` is in `l2`
  - for each `y` in `l2`, check if `x == y`

In [1]:
```
def findcommon(l1,l2):
    commonlist = []
    for x in l1:
        for y in l2:
            if x == y:
                commonlist.append(x)
    return(commonlist)
```

In [2]:
```
l1 = [1,2,3,4]
l2 = [3,4,5,6]
findcommon(l1,l2)
```

Out[2]:
```
[3, 4]
```

- Our function will list repetitions multiple times

In [3]:
```
l1 = [1,2,3,4]
l2 = [3,4,5,3]
findcommon(l1,l2)
```

Out[3]:
```
[3, 3, 4]
```

In [4]:
```
l1 = [1,2,3,4,3]
l2 = [3,4,5,3]
findcommon(l1,l2)
```

Out[4]:
```
[3, 3, 4, 3, 3]
```

- Nested loops can be expensive
- 10^8 operations take about 10 seconds in Python
- Compare the running time of the following nested loops

In [5]:
```
for i in range(1000):
    for j in range(1000):
        x = i + j
print("Done")
```

```
Done
```

In [6]:
```
for i in range(10000):
    for j in range(10000):
        x = i + j
print("Done")
```

```
Done
```

- Can we use the same idea to check if l has duplicates?
- Nested loop over positions in the list rather than values of the list
- Be careful to generate each pair of positions `(i,j)` only once, inner loop starts from `i+1`

In [7]:
```
def duplicates(l):
    for i in range(len(l)):
        for j in range(i+1,len(l)):
            if l[i] == l[j]:
                return(True)
    return(False)
```

In [8]:
```
duplicates([1,2,3])
```

Out[8]:
```
False
```

In [9]:
```
duplicates([1,2,1])
```

Out[9]:
```
True
```

In [10]:
```
def duplicatelist(l):
    dlist = []
    for i in range(len(l)):
        for j in range(i+1,len(l)):
            if l[i] == l[j]:
                dlist.append(l[i])
    return(dlist)
```

In [11]:
```
duplicatelist([1,2,1])
```

Out[11]:
```
[1]
```

In [12]:
```
duplicatelist([1,2,1,3,1])
```

Out[12]:
```
[1, 1, 1]
```

- `x in l` returns `True` if `x` is an element of `l`
- Note that this is implicitly a loop running over all elements in `l`

In [13]:
```
1 in [1,2,3,4]
```

Out[13]:
```
True
```

In [14]:
```
def listcommon2(l1,l2):
    for x in l1:
        if x in l2:
            return(True)
    return(False)
```

if-elif-else

- `sgn(x) = -1` if `x` is negative, `0` if `x` is `0`, `1` if `x` is positive
- Nested `if`, indentation increases
- `if`, `elif` ... `else`

In [15]:
```
def sgn1(v):
    if v > 0:              # First check positive or not
        return(1)
    else:
        if v < 0:          # Then check zero or negative
            return(-1)
        else:
            return(0)
```

In [16]:
```
def sgn2(v):
    if v >= 0:             # First check non-negative or not
        if v > 0:          # Then check positive or zero
            return(1)
        else:
            return(0)
    else:
        return(-1)
```

- Top level 3 way test using `if-elif`

In [17]:
```
def sgn(v):
    if v > 0:
        return(1)
    elif v == 0:
        return(0)
    else:
        return(-1)
```

In [18]:
```
for i in range(-1,2):
    print(sgn(i))
```

```
-1
0
1
```

- For clarity, test each condition explicitly
- As with normal `if`, `else` is optional

In [19]:
```
def sgn3(v):
    if v > 0:
        return(1)
    elif v == 0:
        return(0)
    elif v < 0:
        return(-1)
```

Slice of list

- sublist from position `i` to position `j`
- `l[i:j]` is `[l[i],l[i+1],...,l[j-1]]`
- If `j <= i`, result is empty

In [20]:
```
l = list(range(100))
```

In [21]:
```
l[2:20]
```

Out[21]:
```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [22]:
```
l[20:20],l[23:22]
```

Out[22]:
```
([], [])
```

- Defaults for starting and ending index are `0` and `len(l)`
- Leaving both endpoints blank gives a "full slice" `l[0:len(l)]`

In [23]:
```
l[:20],l[84:]
```

Out[23]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

In [24]:
```
l[:]
```

Out[24]: [0,
1,
2,
3,
4,
5,
6,
7,
8,
9,
10,
11,
12,
13,
14,
15,
16,
17,
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99]

- Like range, can provide a step in a slice
- Step can be negative

In [25]:
```
l[2:20:2]
```

Out[25]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [26]:
```
l[20:1:-1]
```

Out[26]:
```
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
```

In [27]:
```
l[-175:3000]
```

Out[27]:
```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 26,
 27,
 28,
 29,
 30,
 31,
 32,
 33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
 43,
 44,
 45,
 46,
 47,
 48,
 49,
 50,
 51,
 52,
 53,
 54,
 55,
 56,
 57,
 58,
 59,
 60,
 61,
 62,
 63,
 64,
 65,
 66,
 67,
 68,
 69,
 70,
 71,
 72,
 73,
 74,
 75,
 76,
 77,
 78,
 79,
 80,
 81,
 82,
 83,
 84,
 85,
 86,
 87,
 88,
 89,
 90,
 91,
 92,
 93,
 94,
 95,
 96,
 97,
 98,
 99]
```

In [28]:
```
def duplicate3(l):
    duplist = []
    for i in range(len(l)):
        duplist = duplist + findcommon(l[i:],l[i+1:])
    return(duplist)
```

In [29]:
```
duplicate3([1,2,1,3,1])
```

Out[29]:
```
[1, 1, 2, 1, 1, 3, 1, 1, 1, 1, 3, 1, 1, 1, 3, 1, 1]
```

Slice update in a list

- Can update a slice in a list
  - `l[i] = v`
  - `l1[i:j] = l2`
  - This can grow or shrink the list

In [30]:
```
l = list(range(20))
```

In [31]:
```
l
```

Out[31]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [32]:
```
l[2] = 22
```

In [33]:
```
l
```

Out[33]:
```
[0, 1, 22, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [34]:
```
l[3:6] = [23,24,25]
```

In [35]:
```
l
```

Out[35]:
```
[0, 1, 22, 23, 24, 25, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [36]:
```
l[11:20] = [31,32,33]
```

In [37]:
```
l
```

Out[37]:
```
[0, 1, 22, 23, 24, 25, 6, 7, 8, 9, 10, 31, 32, 33]
```

In [38]:
```
l[7:10] = [77,88,99,1000]
```

In [39]:
```
l
```

Out[39]:
```
[0, 1, 22, 23, 24, 25, 6, 77, 88, 99, 1000, 10, 31, 32, 33]
```

In [40]:
```
l[0:5:2] = [5000,5001,5002] # Normally not needed!
```

Nested lists

- A list can contain lists as elements
- Use multiple subscripts to extract inner values

In [41]:
```
l = [[0,1,2], [3,4,5], [6,7,8]]
```

In [42]:
```
l
```

Out[42]:
```
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

In [43]:
```
l[1][1]
```

Out[43]:
```
4
```

In [44]:
```
l[2][1] = 99
```

In [45]:
```
l
```

Out[45]:
```
[[0, 1, 2], [3, 4, 5], [6, 99, 8]]
```

Strings

- Text
- Sequence of characters, operations are similar to a list
- But immutable
- Denote a string using single, double or triple quotes

In [46]:
```
name1 = "abracadabra"
name2 = 'jantarmantar'
```

In [47]:
```
name1, name2
```

Out[47]:
```
('abracadabra', 'jantarmantar')
```

- Use double quotes outside if the string contains a single quote
- Use single quotes outside if the string contains a double quote

In [48]:
```
stmt = "That's his book"
sentence = 'He said "hello"'
```

In [49]:
```
stmt, sentence
```

Out[49]:
```
("That's his book", 'He said "hello"')
```

- Use triple (single) quotes if the string contains both single and double quotes
- Use `\'` and `\"` to indicate that the quote should be taken literally and not given a special meaning

In [50]:
```
complicated = '''He said "That's his book"'''
```

In [51]:
```
complicated
```

Out[51]:
```
'He said "That\'s his book"'
```

- Use positions, slices, concatenation etc as for lists
- No separate single character type; a single character is a string of length 1

In [53]:
```
name1, name1[3], name1[2:7], name1[:]
```

Out[53]:
```
('abracadabra', 'a', 'racad', 'abracadabra')
```

In [54]:
```
a = "a"
```

In [55]:
```
l = [1,2,3]
```

In [56]:
```
l[0] == [1]
```

Out[56]:
```
False
```

In [57]:
```
a, a[0]
```

Out[57]:
```
('a', 'a')
```

- Strings are immutable
  - Cannot reassign a single position or a slice
- **Example:** Change `hello` to `helps`

In [58]:
```
s = "hello"
```

In [59]:
```
s[4] = "s"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [59], line 1
----> 1 s[4] = "s"

TypeError: 'str' object does not support item assignment
```

In [60]:
```
s[3:] = "ps"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [60], line 1
----> 1 s[3:] = "ps"

TypeError: 'str' object does not support item assignment
```

- Use slices, concatenation to reconstruct a new string and reassign to the name

In [61]:
```
s = s[0:3] + "ps"
```

In [62]:
```
s
```

Out[62]:
```
'helps'
```

Tuples

- `(x1,x2,x3)` - round brackets, not square

- Immutable sequence (unlike a list)
- Otherwise manipulate using indices, slices etc

In [63]:
```
dpoint = (0,1,3)
```

In [64]:
```
dpoint[2]
```

Out[64]:
```
3
```

In [65]:
```
dpoint[:2]
```

Out[65]:
```
(0, 1)
```

In [66]:
```
dpoint[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [66], line 1
----> 1 dpoint[2] = 5

TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- A list is a collection indexed by position
- A list can be thought of as a function $f : \{0, 1, \ldots, n-1\} \rightarrow \{v_0, v_1, \ldots, v_{n-1}\}$
  - A list maps positions to values
- Generalize this to a function $f : \{x_0, x_1, \ldots, x_{n-1}\} \rightarrow \{v_0, v_1, \ldots, v_{n-1}\}$
  - Instead of positions, index by an abstract *key*
- **dictionary**: maps keys, rather than positions, to values
- Notation:
  - `d = {k1:v1, k2:v2}` , enumerate a dictionary explicitly
  - `d[k1]` , value in dictionary `d1` corresponding to key `k1`
  - `{}` , empty dictionary (`[]` for lists, `()` for tuples)
  - Keys must be immutable values

In [67]:
```
marks = {"A":3, "B":9}
```

In [68]:
```
marks["A"]
```

Out[68]:
```
3
```

In [69]:
```
marks["C"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In [69], line 1
----> 1 marks["C"]

KeyError: 'C'
```

- Accessing a non-existent key results in `KeyError`
- Analogous to `IndexError` for invalid position in a list

In [70]:
```
marks["C"] = 17
```

In [71]:
```
marks["C"]
```

Out[71]:
```
17
```

- Assigning to non-existent key creates a new key-value pair
  - Unlike lists, where we cannot create a new position by assigning outside the current list
- `d[k] = v` creates `k` if it does not exist, updates the value at `d[k]` if it does exist

- Any immutable value can be a key
  - List cannot be used as a key
- No requirement that all keys (or values) have a uniform type

- Value at a key can be a list
- Use multiple subscripting to access inner components
- Similar to nested lists
- Likewise, value can be a nested dictionary

In [72]:
```
marks = {"A":{"Physics":75,"Maths":88}}
```

In [73]:
```
marks["A"]["Maths"]
```

Out[73]:
```
88
```

Operating on dictionaries

- How do we run through all entries in a dictionary - the equivalent of `for x in l`?
- `d.keys()`, `d.values()` generate sequences corresponding to the keys and values of `d`, respectively
- Like `range()` these are not directly lists, use `list(d.keys())` if you want a list

- Shortcut, can omit `.keys()` when iterating

In [74]:
```
marks
```

Out[74]:
```
{'A': {'Physics': 75, 'Maths': 88}}
```

In [75]:
```
for k in marks.keys():
    print(marks[k])
```

```
{'Physics': 75, 'Maths': 88}
```

In [76]:
```
for k in marks:
    print(k)
```

```
A
```

In [77]:
```
for k in marks.values():
    print(k)
```

```
{'Physics': 75, 'Maths': 88}
```

In [78]:
```
"C" in marks.keys()
```

Out[78]:
```
False
```

In [79]:
```
"C" in marks
```

Out[79]:
```
False
```

In [80]:
```
"C" in marks.values()
```

Out[80]:
```
False
```