Chapter 9.2

Handling Lists

F. Duveau

16/12/11

Objectives of the session: Learning how to extract elements from a list

Creating quickly a list of integers using **`range()`**

Converting between lists and strings

Modifying elements in a pre-existing list

Checking the contents of lists

Tools: Everything will be done with the Python interpreter in the Terminal

```
lucy$ python
>>>
```

Python lists present several specificities compared to other programing languages

Generalities about Python lists

Lists are variables containing collections of values (strings, numbers, lists or mixtures)

Very convenient to handle a large amount of data in a single variable

Contrary to dictionaries, lists are ordered:

- Each element is defined by its position within the list (not by a key)
- There cannot be empty positions within list

Brackets [] are used both for assignation and index specification:

```
>>> MyList = ['a','b','c','d','e']
>>> MyList[0]
['a']
```

Indexing lists

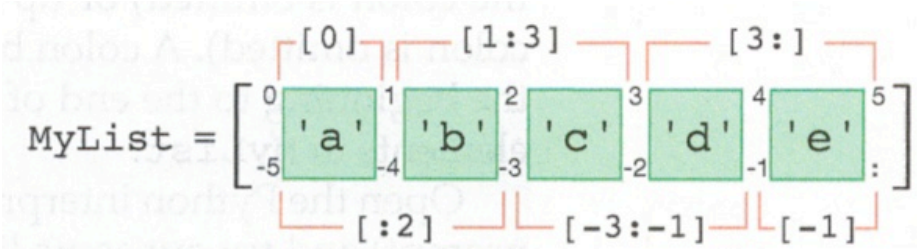

FIGURE 9.2 Numerical ways to think about indexing list elements

The first position in a list is always [0], not [1]

The first colon « **:** » within brackets is used to specify a range of position(s) within a list

```
lucy$ python
>>> MyList = ['a','b','c','d','e']
>>> MyList[:]
['a', 'b', 'c', 'd', 'e']
```

```
>>> MyList[:3]
['a', 'b', 'c']
```

```
>>> MyList[2:]
['c', 'd', 'e']
```

Indexing lists

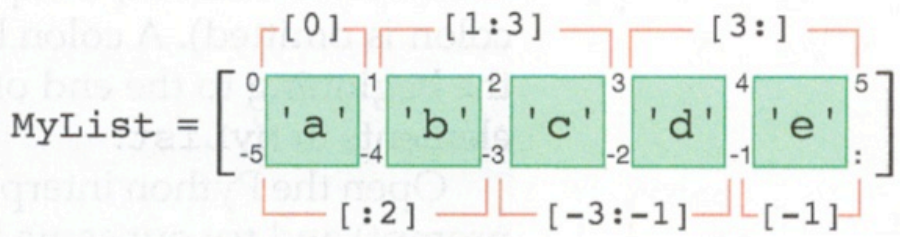

FIGURE 9.2 Numerical ways to think about indexing list elements

Negative values are useful to count from the end of lists of unknown size:

```
>>> MyList[-2:]
['d', 'e']
>>> MyList[:-2]
['a', 'b', 'c']
```

Indexing lists

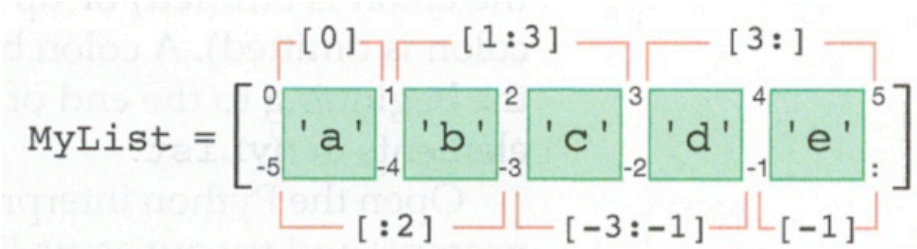

FIGURE 9.2 Numerical ways to think about indexing list elements

A second colon can be used to indicate the **step size** and the **order** of the selection:

```
>>> MyList[0:5:1] ← same as [0:5]
['a', 'b', 'c', 'd', 'e']
>>> MyList[0:5:2]
['a', 'c', 'e']
>>> MyList[::2]
['a', 'c', 'e']
```

```
>>> MyList[::-1]
['e', 'd', 'c', 'b', 'a']
>>> MyList[::-2]
['e', 'c', 'a']
```

Unpacking lists

```
i,j = MyList[:2]
```

Assign the first element of `MyList` to `i` and
the second element to `j`

The number of elements extracted from the list has to be equal to the number of receiving variables

`range()` function to define a list of integers

The range is specified using a comma, not a colon

The range starts from the first parameter and ends just **before** the last parameter

```
>>> RangeList = range(0,6)
>>> RangeList
[0, 1, 2, 3, 4, 5]
```

A third parameter can be used to specify the step size and order of the range:

```
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> range(10,0)   ← A lower limit of 10 doesn't work with 0 as the upper
[]
>>> range(0,10,-1) ← Likewise, stepping backward from 0 to 10 doesn't work
[]
>>> range(10,0,-1) ← Now you're talking
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> range(-5,-11,-1)
[-5, -6, -7, -8, -9, -10]
```

Creating a range of letters in alphabetical order

```
ord('A')
chr(65)
```

Return 65

Return A

65 is the ASCII number corresponding to letter A

Exercise: Print a list of labels corresponding to the wells of a 96-well plate (A1, A2, …, H12)

```
#! /usr/bin/env python
for Let in range(65,73): ← Step through character number 65 to 72
  for Num in range(1,13): ← For each letter, step through numbers 1 to 12
    print chr(Let) + str(Num)
```

(ord('A'),ord('I')) works too

Creating a range of letters in alphabetical order

By default, the `print` command includes an end-of-line character

Adding a comma after a `print` command suppresses this end-of-line

Exercise: Try to display the previous list as a real 96 well-plate (with rows and columns)

```
#! /usr/bin/env python
for Let in range(65,73):
   for Num in range(1,13):
     print chr(Let) + str(Num),   ← The comma is important
   print   ← Prints a line end once for each letter, after 12 numbers have passed
```

The output of this modified script will be eight lines of twelve elements:

```
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12
...
H1 H2 H3 H4 H5 H6 H7 H8 H9 H10 H11 H12
```

Comparison between lists and strings

Strings behave as lists of characters:
Both can be combined using +, sorted and iterated within a `for` loop

BUT

Lists can be modified and not strings

```
lucy$ python
>>> SeqString = 'ACGTA'
>>> SeqList = ['A', 'C', 'G', 'T', 'A']
>>> SeqString[3]
'T'
>>> SeqList[3]
'T'
>>> SeqString[3]='U'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> SeqList[3]='U'
>>> SeqList
['A', 'C', 'G', 'U', 'A']
```

Converting between lists and strings

String -> List `list()` function

```
>>> MyString = 'abcdefg'
>>> MyList = list(MyString)
>>> MyList
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

List -> String `.join()` method

`.join()` is a string method that takes a list argument

```
>>> MyList = ['ab', 'cde', 'fghi']
>>> ''.join(MyList)
'abcdefghi'
>>> '\t'.join(MyList)
'ab\tcde\tfghi'
>>> ' '.join(MyList)
'ab cde fghi'
```

Modifying existing lists

Adding elements

```
x = ['A','B']
```

~~`x[2] = 'C'`~~

Use `.append()` method:

```
x.append('C')
```

It is possible to directly insert new elements within a list:

```
>>> MyList=['a','e']   ← Define a list with two elements
>>> MyList[1:1] = ['b','c','d']   ← Insert into position 1
>>> MyList
['a','b','c','d','e']
```

Removing elements

Use `del()` function or reassign an empty list:

```
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> MyList[2:5]=[]
>>> MyList
[10, 11, 15, 16, 17, 18, 19]
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> del(MyList[2:5])
>>> MyList
[10, 11, 15, 16, 17, 18, 19]
```

Testing the contents of a list

Use the `in` operator:

```
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> 11 in MyList
True
>>> 21 in MyList
False
```

Sorting lists

`.sort()` method to sort in place without changing the list

```
>>> MyList = [4,3,6,5,2,9,0,8,1,7]
>>> MyList.sort()    ← You don't have to assign the output to a new variable
>>> MyList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`sorted()` function allows to save the sorted list in a new variable

```
>>> MyList = [4,3,6,5,2,9,0,8,1,7]
>>> NewList=sorted(MyList)
>>> MyList    ← The original list is unchanged
[4, 3, 6, 5, 2, 9, 0, 8, 1, 7]
>>> NewList ← The sorted list has been placed here
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Identifying unique elements in lists and strings

The `set()` function returns all individual elements contained in a list:

```
>>> Colors = ['red','red','blue','green','blue']
>>> list(set(Colors))
['blue', 'green', 'red']
>>> DNASeq = 'ATG-TCTCATTCAAAG-CA'
>>> list(set(DNASeq))
['A', 'C', '-', 'T', 'G']
```

The output of the `set()` function is not a list

List comprehension

How can we easily apply a same modification to all elements of a list ?

Direct use of operators or methods (such as `MyList.upper()`) does not work

```
>>> MyList = range(0,5)
>>> MyList
[0, 1, 2, 3, 4]
>>> MyList * 2
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

One solution is to write a `for` loop through each element of the list:

```
>>> Values = range(1,11)
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = []
>>> for Value in Values:
...     Squares.append(Value**2)
...
>>> Squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List comprehension

How can we easily apply a same modification to all elements of a list ?

A list comprehension is a 1-line `for` loop specifically designed to modify lists

The operation to be done in the loop is written before the `for` command

```
>>> Values = range(1,11)
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = [Element**2 for Element in Values]
>>> Squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Useful to extract columns of data from 2D arrays of strings:

```
>>> GeneList = ['ATTCAGAAT','TGTGAAAGT','TGTATCGCG','ATGTCTCTA']
>>> FirstCodons = [ Seq[0:3] for Seq in GeneList ]
>>> FirstCodons
['ATT', 'TGT', 'TGT', 'ATG']
```

```
>>> Linker='GAATTC'
>>> Start = [(Linker + Seq[0:3]) for Seq in GeneList ]
>>> Start
['GAATTCATT', 'GAATTCTGT', 'GAATTCTGT', 'GAATTCATG']
```

List comprehension

Many other operations can be done using list comprehension:

```
>>> [ Seq.count('A') for Seq in GeneList ]
[4, 3, 1, 2]
```

Although you can convert a string to a list of characters using the `list()` function, it is sometimes hard to go from a list of numbers `[1,2,3]` to the equivalent strings `['1','2','3']`. You can't just use `str(ListOfIntegers)`. List comprehension again comes to the rescue:

```
>>> [ str(N) for N in range(0,10) ]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Things to know about copying / modifying variables

In Python, copying a variable does not create a new variable targeted to a new place in computer memory

Instead, it creates a new name that points to the same place as the copied variable in computer memory

```
x=5
y=x
```
➡ x and y point to the same place in memory with value 5

```
x=8
```
➡ A new x variable is created with value 8, so y remains unchanged

It becomes more tricky with lists that can be modified without creation of a new variable

```
A=[1,2,3]
B=A
```
➡ A and B point to the same place in memory containing the list of integers [1,2,3]

```
A[1]=4
```
➡ Both A and B are changed to [1,4,3]

Good way to copy a list: `B=A[:]` ➡ Creates a new variable B at a different place in memory