Lecture 2
Sequences

Lecturer: Pieter De Bleser

Bioinformatics Core Facility, IRC

Slides derived from: I. Holmes, Department of Statistics, University of Oxford; M. Schroeder, M. Adasme, A. Henschel, Biotechnology Center, TU Dresden; S. Spielman, CCBB, University of Texas; **Jehan-François Pâris, jfparis@uh.edu**

Overview

- **Types** of sequences and their properties
  - Lists, Tuples, Strings, Iterators
- Building, accessing and modifying sequences
- **List comprehensions**
- **File operations**

Lists

Lists

Lists are sequences of items separated by commas enclosed in square brackets ("[…]")

Examples

```
[ 1, 2, 3, 4, 5]
['Annafrid', 'Bjorn', 'Charlie', Danny']
['Agnetha', 'Benny', [100, 90, 70]]
```

Usages

- Group together similar objects
  - Lists of people, courses, exams
  - Vectors and matrices of algebra and physics
- Store records
  - Contact list entry
    - ['Donald', 666-555-1234, 'donald@whitehouse.org']
  - Anything else

Operations

- Access the elements of a list:
  - Individual elements
  - Slices
- Modify its elements
  - Insert elements
  - Anywhere
- Remove elements

Lists

```
>>> mylist = [11, 12, 13, 14, 'done']
>>> mylist
[11, 12, 13, 14, 'done']
>>> print(mylist)
[11, 12, 13, 14, 'done']
>>> mylist[0]
11
>>> mylist[0:1]
[11]
```

We can access individual elements

Two observations
1. A ***list slice*** is a list
2. mylist[0:1] starts with **list[0]** but stops **before mylist [1]**

More list slices

```
>>> mylist[0:2]
[11, 12]
```

Includes **mylist[0]** and **mylist[1]**

```
>>> mylist[0:]
[11, 12, 13, 14, 'done']
```

The whole list

```
>>> mylist[1:]
[12, 13, 14, 'done']
```

```
>>> mylist[-1:]
['done']
```

A ***list slice*** is a list

```
>>> mylist[-1]
'done'
```

Not the same thing!

List of lists

```
>>> a = [[1, 2], [3, 1]]
>>> a
[[1, 2], [3, 1]]
>>> a[0]
[1, 2]
>>> a[1][1]
1
```

Can be used to represent matrices

Sorting lists

```
>>> mylist = [11, 12, 13, 14, 'done']
>>> mylist.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str'
and 'int'
```

Cannot compare apples and oranges

Sorting lists of strings

In-place sort

```
>>> namelist = ['Alpha', 'Gamma', 'Beta']
>>> namelist.sort()
>>> print(namelist)
['Alpha', 'Beta', 'Gamma']
```

namelist.sort() is a *method* applied to the list namelist

Sorting lists of numbers

In-place sort

```
>>> newlist = [0, -1, +1, -2, +2]
>>> newlist.sort()
>>> print(newlist)
[-2, -1, 0, 1, 2]
```

Sorting into a new list

```
>>> newlist = [0, -1, +1, -2, +2]
>>> sortedlist= sorted(newlist)
>>> print(newlist)
[0, -1, 1, -2, 2]
>>> print(sortedlist)
[-2, -1, 0, 1, 2]
```

sorted(...) is a conventional Python function that returns a new list

Modifying the elements of a list

```
>>> mylist = [11, 12, 13, 14, 'done']
>>> mylist[-1] = 'finished'
>>> mylist
[11, 12, 13, 14, 'finished']
>>> mylist[0:4] = ['XI', 'XII', 'XIII', 'XIV']
>>> mylist
['XI', 'XII', 'XIII', 'XIV', 'finished']
```

Lists of lists (I)

```
>>> listoflists = [[17.5, "1306"], [13, "6360"]]
>>> listoflists.sort()
>>> print(listoflists)
[[13, '6360'], [17.5, '1306']]
>>> listoflists[0]
[13, '6360']
>>> listoflists[1]
[17.5, '1306']
```

Lists of lists (II)

```
>>> listoflists[0][0]
13
>>> listoflists[1][1]
'1306'
>>> listoflists[0][1] ='6360 quiz'
>>> listoflists
[[13, '6360 quiz'], [17.5, '1306']]
```

Adding elements to a list

```
>>> mylist = [11, 12, 13, 14, 'finished']
>>> mylist.append('Not yet!')
>>> mylist
[11, 12, 13, 14, 'finished', 'Not yet!']
```

Adding elements to a list (I)

```
>>> mylist = [11, 12, 13, 14, 'finished']
>>> mylist.append('Not yet!')
>>> mylist
[11, 12, 13, 14, 'finished', 'Not yet!']
```

Adding elements to a list (II)

```
>>> listoflists = [[13, '6360 quiz'], [17.5, '1306']]
>>> listoflists.append([15.3, 'ABET'])
>>> listoflists
[[13, '6360 quiz'], [17.5, '1306'], [15.3, 'ABET']]
```

Appending means adding at the end.

Adding elements inside a list (I)

```
>>> mylist = [11, 12, 13, 14, 'finished']
>>> mylist.insert(0, 10)
>>> mylist
[10, 11, 12, 13, 14, 'finished']
>>> mylist.insert(5, 15)
>>> mylist
[10, 11, 12, 13, 14, 15, 'finished']
```

Adding elements inside a list (II)

```
>>> mylist = [11, 12, 13, 14, 'finished']
>>> mylist.insert(0, 10)
>>> mylist
[10, 11, 12, 13, 14, 'finished']
>>> mylist.insert(5, 15)
>>> mylist
[10, 11, 12, 13, 14, 15, 'finished']
```

Adding elements inside a list (III)

`mylist.insert(index, item)`

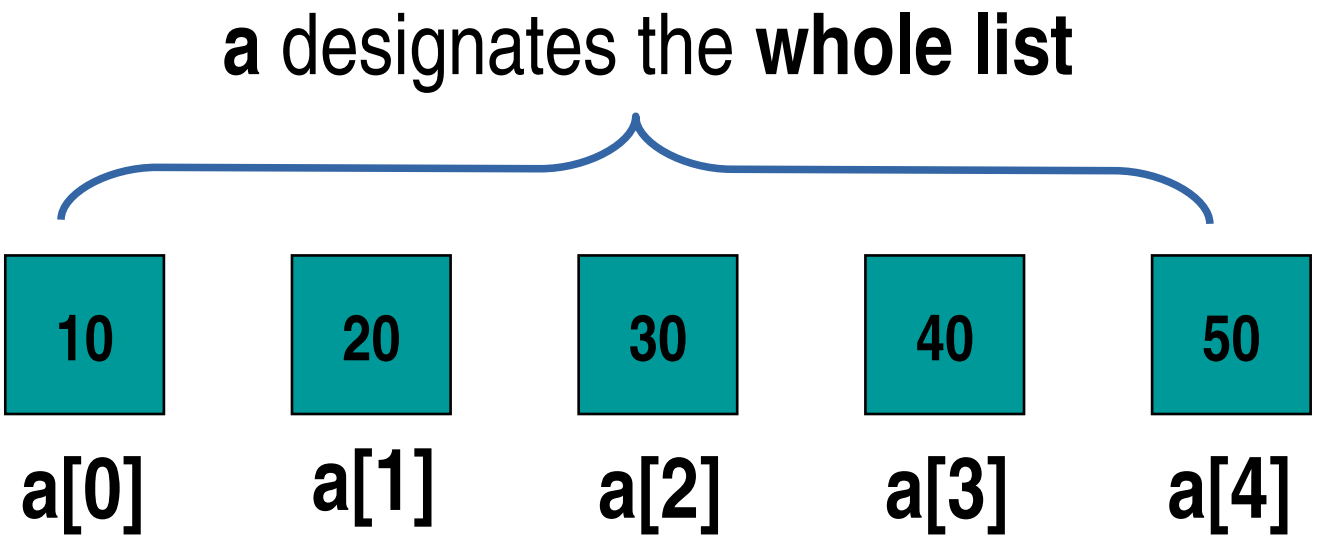
index specifies element before which the new **item should be inserted**

`mylist.insert(0, item)` inserts the new item before the first element

`mylist.insert(1, item)` inserts the new item before the second element and after the first one

Example (I)

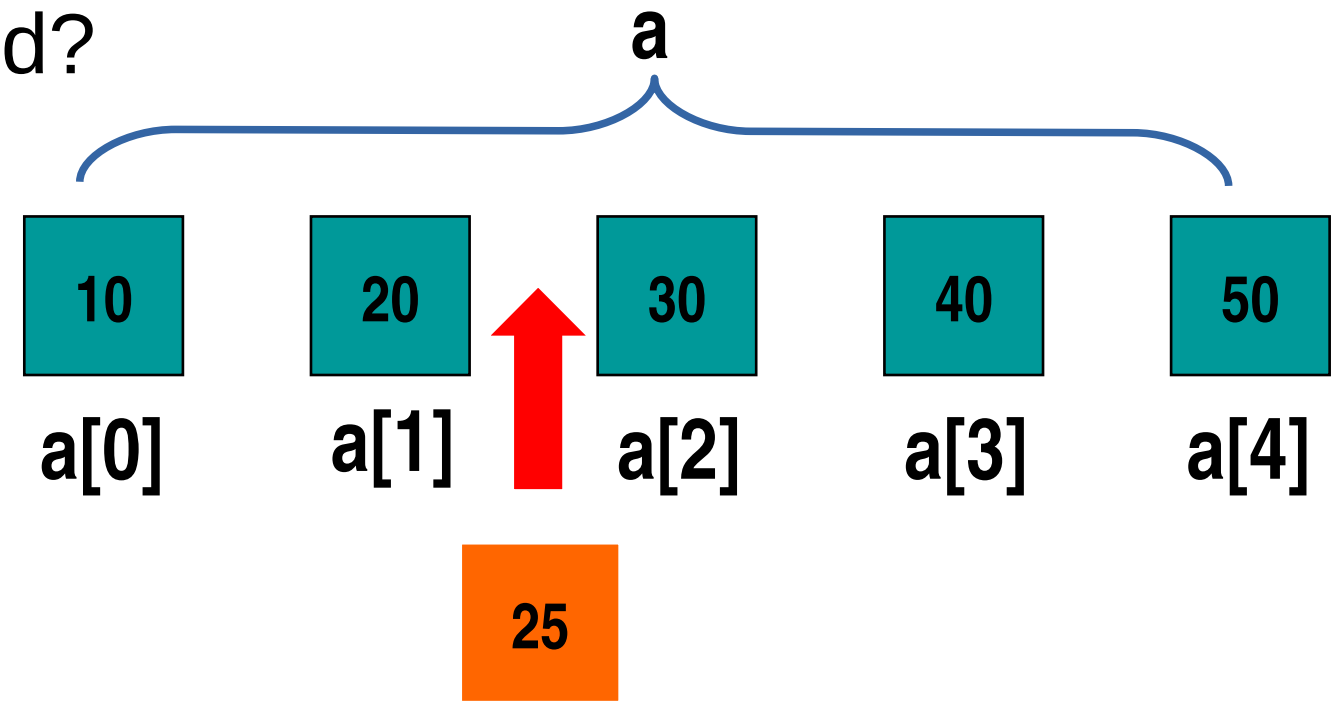
a = [10, 20, 30, 40, 50]



Example (II)

a = [10, 20, 30, 40, 50]

Where to insert 25 and keep the list sorted?



Example (III)

We do

`a.insert(2, 25)`

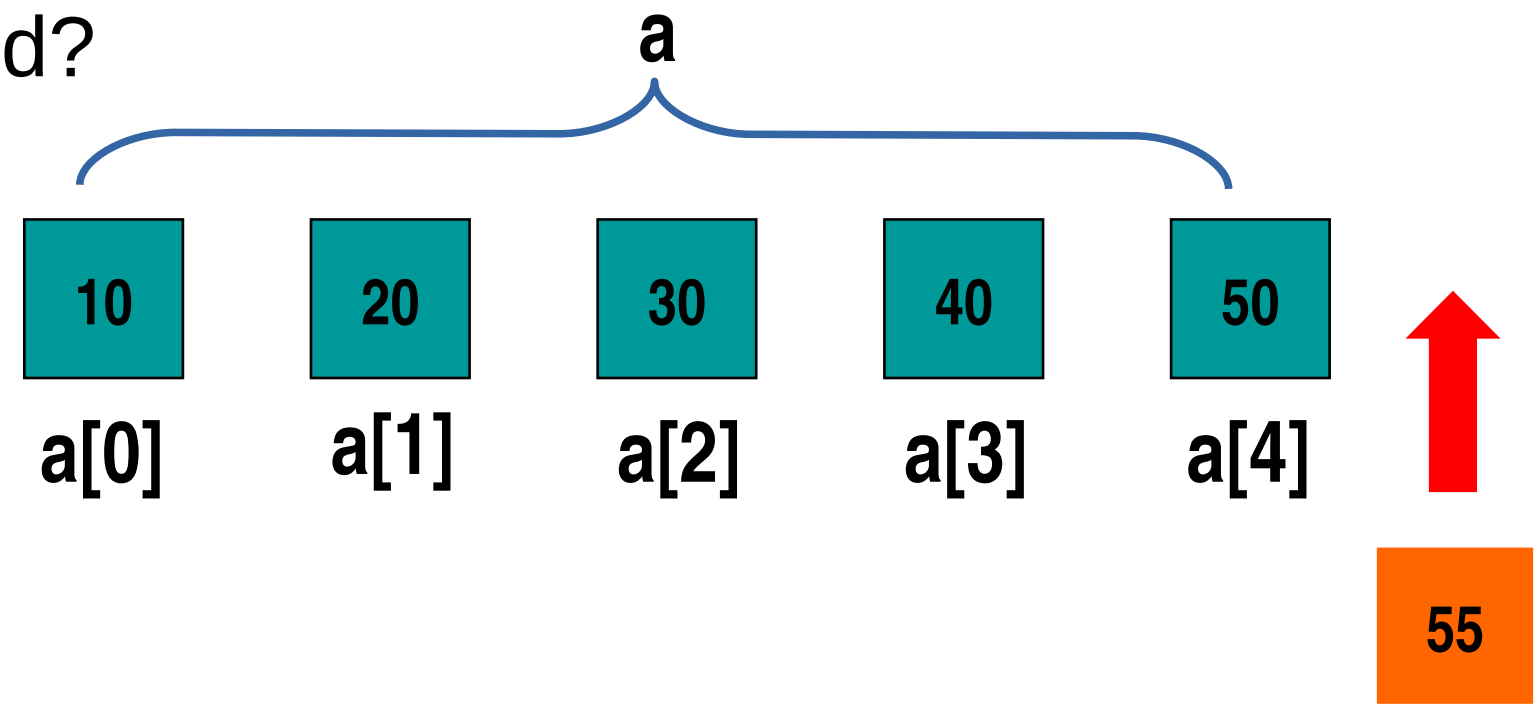
after **a[1]** and before **a[2]**

```
>>> a = [10, 20, 30, 40, 50]
>>> a.insert(2,25)
>>> a
[10, 20, 25, 30, 40, 50]
```

Example (IV)

a = [10, 20, 30, 40, 50]

Where to insert 55 and keep the list sorted?



Example (V)

- We must insert

  After **a[4]**

  Before no other element

- We act ***as if* a[5] *existed***

  `a.insert(5, 55)`

  It works!

  Same as **`a.append(55)`**

Let us check

```
>>> a = [10, 20, 30, 40, 50]
>>> a.insert(5, 55)
>>> a
[10, 20, 30, 40, 50, 55]
```

Let us make it a bit easier

- **`len(list)`**

  returns the **length** of a list

  Same as **number of its elements**

  Equal to **index of last element plus one**

- We could have written

  `a.insert(len(a), 55)`

  Same as **`a.append(55)`**

Removing items

- One by one
- **`thislist.pop(i)`**

  removes **`thislist[i]`** from **thislist**

  returns the ***removed element***
- **`thislist.pop( )`**

  removes **the last element** from **thislist**

  returns the ***removed element***

Examples (I)

```
>>> mylist = [11, 22, 33, 44, 55, 66]
>>> mylist.pop(0)
11
>>> mylist
[22, 33, 44, 55, 66]
```

Examples (II)

```
>>> mylist.pop()
66
>>> mylist
[22, 33, 44, 55]
>>> mylist.pop(2)
44
>>> mylist
[22, 33, 55]
```

Sum: one more useful function

```
>>> list = [10, 20 ,20]
>>> sum(list)
50
>>> list = ['Oliver', ' and ', 'Laurel']
>>> sum(list)
```

Does **not** work!

Would get same results with ***tuples***

Initializing lists

Use ***list comprehensions***

```
>>> [ 0 for n in range(0, 9)]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [n for n in range (1,11)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [2*n+1 for n in range(0, 6)]
[1, 3, 5, 7, 9, 11]
```

Warning!

- The **`for n`** clause is ***essential***

- `[0 in range(0, 10)]`
  `[True]`

  Because 0 is in range(0, 10)

More comprehensions

```
>>> [ c for  c in 'PY4BIO 2019']
['P', 'Y', '4', 'B', 'I', 'O', ' ', '2', '0', '1', '9']
>>> names = ['agnetha', 'annafrid', 'bjorn']
>>> cap_names =[n.capitalize() for n in names]
>>> names
['agnetha', 'annafrid', 'bjorn']
>>> cap_names
['Agnetha', 'Annafrid', 'Bjorn']
```

An equivalence

- ```
  [2*n+1 for n in range(0, 6)]
  [1, 3, 5, 7, 9, 11]
  ```

is same as

- ```
  a = []
  for n in range(0,6) :
    a.append(2*n + 1)
  ```

Filtered comprehensions

```
>>> a = [11, 22, 33, 44, 55]
>>> b = [ n for n in a if n%2 == 0]
>>> b
[22, 44]
>>> c = [n//11 for n in a if n > 20]
>>> c
[2, 3, 4, 5]
```

// for “floor” division (rounds down to nearest whole number)

A very powerful tool !

The for statement (I)

- **for i in range(low, high):**

  Repeats **high – low** times

  For all integer values of **i** between **low** and **hi – 1**

  low ≤ i < hi

Example

```
>>> for i in range(1, 5) :
...     print ('Iteration %s' %i)
...
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

5 is NOT in range(1, 5)

Observation

- If you want to execute a loop ***n times***, use

  for i in range(0,5)

  for i in range (1, 6)

  for i in range (2, 7)

  …

The for statement (II)

- **for i in a :**

  a must be a **list**

  Repeats **len(a)** times

  For all elements of a

  for names in [‘Alice’, ‘Bob’, ‘Carol’]

Example (I)

```
>>> names = ['Alice', 'Bob', 'Carol']
>>> for name in names :
   print('Hi ' + name + '!')

Hi Alice!
Hi Bob!
Hi Carol!
```

Membership

- Can test membership in a list

  a in alist returns

  True is a is in **alist** and

  False otherwise

- Will use it in if statements

  if a in alist :

Copying/Saving a list

```
>>> a = ["Alice", "Bob", "Carol"]
>>> saved = a
>>> a.pop(0)
'Alice'
>>> a
['Bob', 'Carol']
>>> saved
['Bob', 'Carol']
```

We did not save anything!

Mutable and immutable quantities

- By default, Python quantities are ***immutable***

  Each time you modify them, you create a new value

- ***Expensive solution that works as we expect it***

Mutable and immutable quantities

- Python lists are ***mutable***

  They can be modified in place

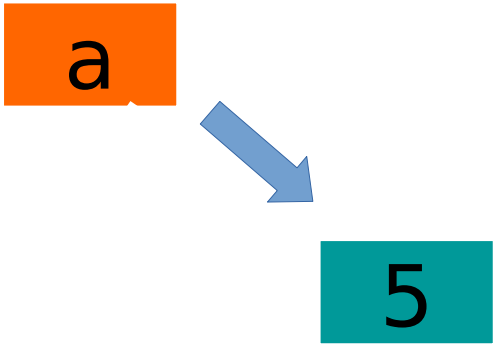
  a* = [1, 5, 3] *can be sorted without making a new copy of the list

- ***Much cheaper solution***

  Can cause surprises!

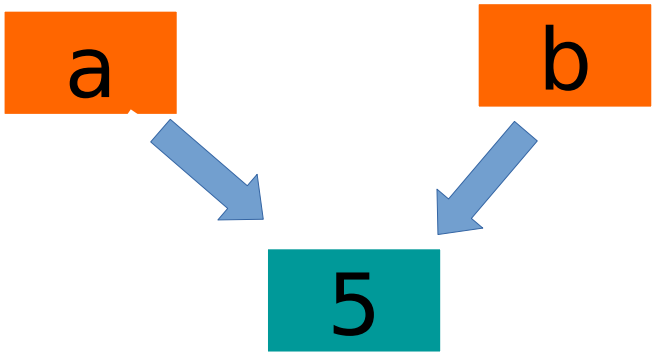
How Python implements variables

- A Python variable contains the address of its current value
- **a = 5**



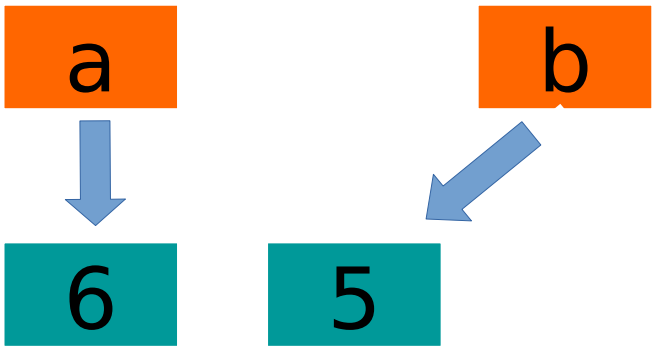
How Python implements variables

- A Python variable contains the address of its current value
- **a = 5**
  b = a



How Python implements variables

- A Python variable contains the address of its current value
- **a = 5**
  b = a
  a = a +1



How Python implements variables

- This work as we expected

  We ***saved*** the old value of a into b before modifying it

- People write

```
a = 5      # initial value
b = a      # save initial value
a = a +1   # increment
```

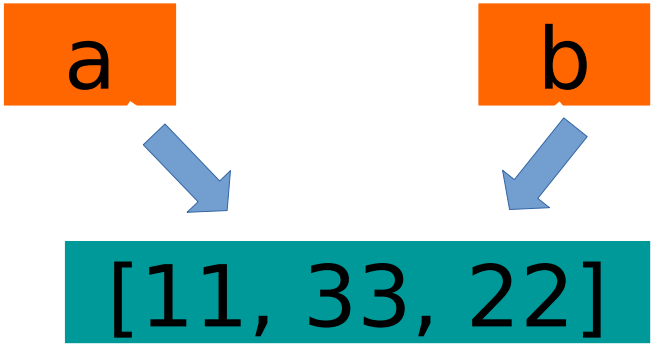
A big surprise

```
>>> a = [11, 33, 22]
>>> b = a
>>> b
[11, 33, 22]
>>> a.sort()
>>> a
[11, 22, 33]
>>> b
[11, 22, 33]
```

The old value of a was never saved!

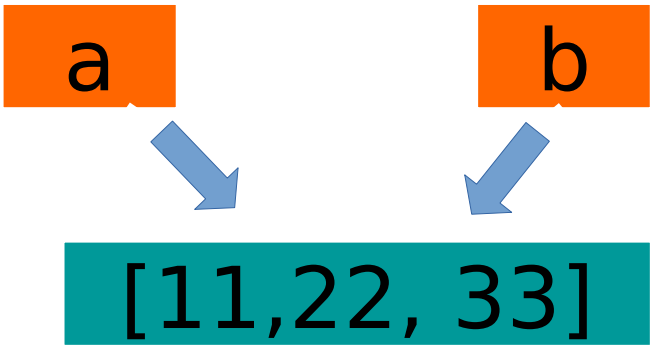
What happened (I)

```
>>> a = [11, 33, 22]
>>> b = a
>>> b
[11, 33, 22]
```



What happened (II)

```
>>> a = [11, 33, 22]
>>> b = a
>>> b
[11, 33, 22]
>>> a.sort()
>>> a
[11, 22, 33]
>>> b
[11, 22, 33]
```



Why this mess?

- Making lists immutable would have made Python much slower

- Conflict between ease of use and efficiency

  This time efficiency won!

  Because efficiency penalty would have been very big

How to copy a list

Copy a slice containing the whole list

```
>>> a = [11, 33, 22]
>>> b = a[0:len(a)]
>>> a.sort()
>>> a
[11, 22, 33]
>>> b
[11, 33, 22]
```

Or: `b=a[:]`

Silly but it works!

Lists vs Strings: Similarities

In Python strings and lists are both sequences that can be indexed. In fact, all of the built-in string operations that we discussed previously are sequence operations and can also be applied to lists:

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[ ]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration

Lists vs Strings: Differences

- The items in a list can be any data type, including instances of programmer-defined classes. Strings, obviously, are always sequences of characters.
- Second, lists are ***mutable***. *That means that the contents of a list can be modified. Strings cannot be changed* "in place."

```
>>> my_list = [34,26,15,10]
>>> my_list[2]
15
>>> my_list[2]=0
>>> my_list
[34, 26, 0, 10]
>>> my_string="Hello World"
>>> my_string[2]
'l'
>>> my_string[2]='z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

List operations

- Python lists are dynamic. They can grow and shrink on demand. They are also *heterogeneous.* You can mix arbitrary data types in a single list. In a nutshell, *Python lists are mutable sequences of arbitrary objects.* This is very different from arrays in other programming languages.
- A list of identical items can be created using the **repetition operator**.

```
>>> zeroes = [0] * 30
>>> zeroes
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Typically, lists are built up one piece at a time using the **append** method.

```
data = []   # start with empty list

# loop to get numbers
number = input( "Enter a number (<ENTER> to quit) >> " )

while number != "":
    x = float(number)
    data.append( x )
    number = input( "Enter a number (<ENTER> to quit) >> " )

print( 'The list contains following numbers: ', data )
```

```
> python3 list_append.py
Enter a number (<ENTER> to quit) >> 0
Enter a number (<ENTER> to quit) >> 1.1
Enter a number (<ENTER> to quit) >> 2.3
Enter a number (<ENTER> to quit) >> 3.1415
Enter a number (<ENTER> to quit) >>
You entered following numbers:  [0.0, 1.1, 2.3, 3.1415]
```

- List operations: remove elements

```
>>> my_list = [34,26,15,10]
>>> del my_list[1]
>>> my_list
[34, 15, 10]
>>> del my_list[1:3]
>>> my_list
[34]
>>>
```

List operations

Method	Meaning
< `list` >.append(x)	Add element x to end of list.
< `list` >.sort()	Sort the list. A comparison function may be passed as parameter.
< `list` >.reverse()	Reverses the list.
< `list` >.index(x)	Returns index of first occurrence of x.
< `list` >.insert(i,x)	Insert x into list at index i. (Same as `list[i:i] = [x]`)
< `list` >.count(x)	Returns the number of occurrences of x in list.
< `list` >.remove(x)	Deletes the first occurrence of x in list.
< `list` >.pop(i)	Deletes the ith element of the list and returns its value.
x in <*list* >	Checks to see if x is in the list (returns a Boolean).

Lists vs tuples

- Both are sequences
- Lists should be used for storing equal elements
- Tuples should be used for storing different elements
- Lists are larger than tuples (i.e. consume more memory)

Construction (Syntax)

```
mylist = [1,2,3,4]
mylist2 = [‘Apple’, ‘Banana’, ‘Orange’]
```

```
mytuple = (‘sebastian’, ‘m’, 28)
mytuple2 = (‘motif’, ‘ATTCG’, ‘E44’)
```

Accessing Elements

```
mylist[0] → 1
```

```
mytuple[0] → ‘sebastian’
```

Modifying Elements

```
mylist[1] = 5
```

```
mytuple[1] = 5
```

IMMUTABLE!

Adding Elements

```
mylist += [3,2]
```

```
mytuple += (‘phd’,’biotec’)
```

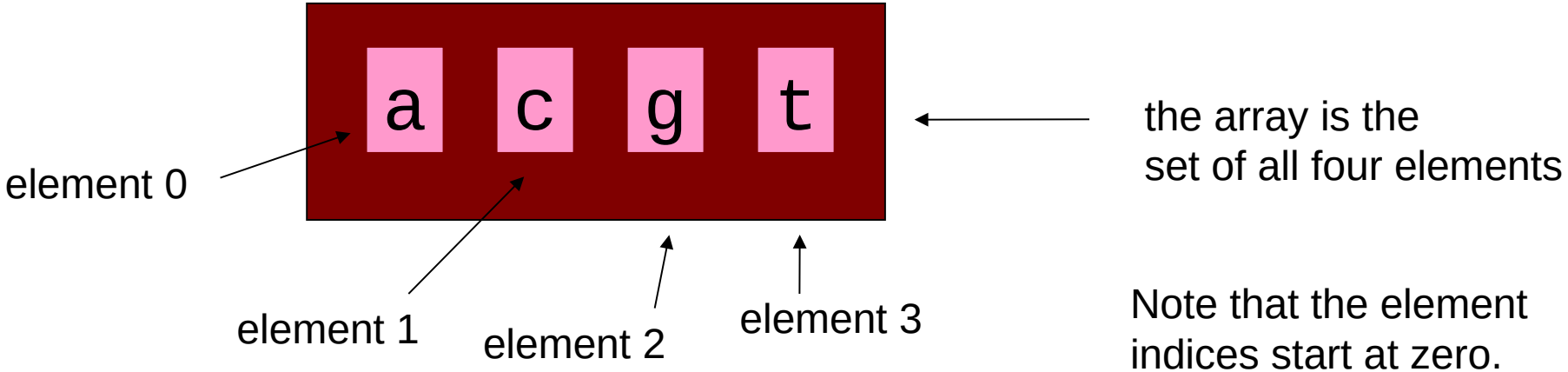
Lists

A *list* is a list of variables

```
nucleotides = ['a', 'c', 'g', 't']
print("Nucleotides: ",nucleotides)
```

```
Nucleotides:  ['a', 'c', 'g', 't']
```

We can think of this as a list with 4 entries



List literals

There are several, equally valid ways to assign an entire array at once.

This is the most common: a commaseparated list, delimited by squared brackets

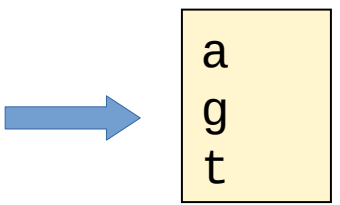
```
a = [1,2,3,4,5]
print("a = ",a)
b = ['a','c','g','t']
print("b = ",b)
c = range(1,6)
print("c = ",c)
d = "a c g t".split()
print("d = ", d)
```

```
a = [1,2,3,4,5]
b = ['a','c','g','t']
c = [1,2,3,4,5]
d = ['a','c','g','t']
```

Accessing lists

To access list elements, use square brackets e.g. x[0] means "element zero of list x"

```
x = ['a', 'c', 'g', 't']
print(x[0])
i = 2
print(x[i])
print(x[-1])
```



```
a
g
t
```

- Element indices start at zero!
- Negative indices refer to elements counting from the end e.g. x[-1] means "last element of list x"

List operations

- You can sort and reverse lists…

```
x = ['a', 't', 'g', 'c']
print("x =",x)
x.sort()
print("x =",x)
x.reverse()
print("x =",x)
```

```
x = ['a', 't', 'g', 'c']
x = ['a', 'c', 'g', 't']
x = ['t', 'g', 'c', 'a']
```

- You can add, delete and count elements

```
nums = [2,2,5,2,6]
nums.append(8)
print(nums)
print(nums.count(2))
nums.remove(5)
print(nums)
```

```
[2,2,5,2,6,8]
3
[2,2,2,6,8]
```

More list operations

Multiplying lists with *

pop removes the last element of a list

append adds an element to the end of a list

concatenating lists with + or +=

Removing the first occurrence of an element

Position of an element

```
>>> x=[1,0]*5
>>> x
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
>>> while 0 in x: print(x.pop())
0 1 0 1 0 1 0 1 0
>>> x
[1]
>>> x.append(2)
>>> x
[1, 2]
>>> x+=x
>>> x
[1, 2, 1, 2]
>>> x.remove(2)
>>> x
[1, 1, 2]
>>> x.index(2)
2
```

Example: Reverse complementing DNA

A common operation due to double-helix symmetry of DNA

Start by making string lower case again.
This is generally good practice

Replace 'a' with 't', 'c' with 'g',
'g' with 'c' and 't' with 'a'
Reverse the list

```
replaced=list("accACgttAGgtct ".lower().
         replace("a","x").replace("t","a").
         replace("x", "t").replace("g","x").
         replace("c","g").replace("x", "c"))
replaced.reverse()
print("".join(replaced))
```

```
agacctaacgtggt
```

for loop revisited

Finding the total of a list of numbers:

for statement loops
through each entry in a list

```
val = [4, 19, 1, 100, 125, 10]
total = 0
for x in val:
    total += x
print total
```

259

```
val = [4, 19, 1, 100, 125, 10]
total = 0
for i in range(len(val)):
total += val[i]
print total
```

259

```
val = [4, 19, 1, 100, 125, 10]
print sum(val)
```

259

Taking a slice of a list

The syntax x[i:j] returns a list containing elements i,i+1,…,j-1 of list x

```
nucleotides = ['a', 'g', 'c', 't']
purines = nucleotides[0:2]
pyrimidines = nucleotides[2:4]
print("Nucleotides:", nucleotides)
print("Purines:", purines)
print("Pyrimidines:", pyrimidines)
```

```
Nucleotides: ['a', 'g', 'c', 't']
Purines: ['a', 'g']
Pyrimidines: ['c', 't']
```

Lists and Strings

- A string can be converted into a list of strings
  - Using the `split` method: string.split(separator)
- A list of strings can be converted into one string
  - Using the `join` method: separator.join(list)

```
sentence = 'This is a complete sentence.'
print(sentence.split())
```

```
['This', 'is', 'a', 'complete', 'sentence.']
```

```
datarow = 'Apples,Bananas,Oranges'
print(datarow.split(','))
```

```
['Apples', 'Bananas', 'Oranges']
```

```
cities=['Antwerp','Brussels','Ghent','New York']
print('-->'.join(cities))
```

```
Antwerp-->Brussels-->Ghent-->New York
```

list comprehensions

- Easy way to **construct** sequences, especially lists
- Often replaces a for loop and an if-construction
- Is used very often in Python
- Syntax: [**expr(var) for var in sequence if condition**]

Ex.: Find the squares of all odd numbers between 1 and 10 → `[1,9,25,49,81]`

Naive construction of list!

```
newlist = []
for x in range(1,11):
    if x % 2:
        newlist.append(x**2)
```

Construction with list comprehension:

```
newlist = [x**2 for x in range(1,11) if x % 2]
```

Examples: List comprehensions

```
sentence = 'I like MySQL but not Python'
print([(w.lower(), len(w)) for w in sentence.split()])
```

```
[('i', 1), ('like', 4), ('mysql', 5), ('but', 3), ('not', 3), ('python', 6)]
```

Constructs a generator (iterator) with the sequence of numbers

```
generator = ((x-2)*3 for x in range(10))
```

Find the sum of all positive integers in my tuple

```
my_numbers = (1,0,-1,6,3,-2,3,4)
my_sum = sum([x for x in my_numbers if x >0])
print('sum: ',my_sum)
```

```
sum:
17
```

Tuples

Tuples

Same as lists but

- Immutable
- Enclosed in parentheses
- A tuple with a single element ***must*** have a comma inside the parentheses:

```
a = (11,)
```

Tuples – Examples

```
>>> my_tuple = (34,26,15,10)
>>> type(my_tuple)
<class 'tuple'>
>>> my_tuple[0]
34
>>> my_tuple[-1]
10
>>> my_tuple[0:1]
(34,)
>>> type(my_tuple[0:1])
<class 'tuple'>
>>> type( (34) )
<class 'int'>
>>> type( [34] )
<class 'list'>
```

The comma is required!

Tuples are immutable

```
>>> my_tuple = (34,26,15,10)
>>> saved=my_tuple
>>> my_tuple+=(44,)
>>> my_tuple
(34, 26, 15, 10, 44)
>>> saved
(34, 26, 15, 10)
```

This will not work!

```
>>> my_tuple+=55
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int")
to tuple
```

Sorting tuples

```
>>> my_tuple = (34,26,15,10)
>>> my_tuple.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
>>> my_tuple=sorted(my_tuple)
>>> my_tuple
[10, 15, 26, 34]
>>> type(my_tuple)
<class 'list'>
```

sorted( ) returns a list!

Most other things work!

```
>>> my_tuple = (34,26,15,10)
>>> len(my_tuple)
4
>>> 26 in my_tuple
True
>>> [i for i in my_tuple]
[34, 26, 15, 10]
```

The reverse does not seem to work at first, but...

```
>>> my_list = [34, 26, 15, 10]
>>> (i for i in my_list)
<generator object <genexpr> at 0x7f78edc3d660>
>>> tuple( (i for i in my_list) )
(34, 26, 15, 10)
```

Converting sequences into tuples

```
>>> alist = [11, 22, 33]
>>> atuple = tuple(alist)
>>> atuple
(11, 22, 33)
>>> newtuple = tuple('Hello World!')
>>> newtuple
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!')
```

Sets

Sets

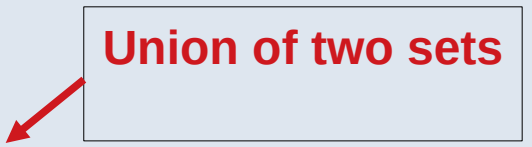
- Identified by ***curly braces***
- **{'Annafrid', 'Bjorn', 'Carol'}**
- **{'George'}** is a ***singleton***
- Can only contain ***unique elements***
- ***Duplicates are eliminated***
- ***Immutable*** like tuples and strings

Sets do not contain duplicates

```
>>> cset = {11, 11, 22}
>>> cset
{11, 22}
```

Sets are immutable

```
>>> aset = {11, 22, 33}
>>> bset = aset
>>> bset
{33, 11, 22}
>>> aset = aset | {55}
>>> aset
{33, 11, 22, 55}
>>> bset
{33, 11, 22}
```



Sets have no order

```
>>> {1, 2, 3, 4, 5, 6, 7}
{1, 2, 3, 4, 5, 6, 7}
>>> {11, 22, 33}
{33, 11, 22}
```

Sets do not support indexing

```
>>> myset = {'Apples', 'Bananas', 'Oranges'}
>>> myset
{'Oranges', 'Bananas', 'Apples'}
>>> myset[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Examples

```
>>> alist = [11, 22, 33, 22, 44]
>>> aset = set(alist)
>>> aset
{33, 11, 44, 22}
>>> aset = aset + {55}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'set' and 'set'
```

Boolean operations on sets

Union of two sets:

```
>>> aset = {11, 22, 33}
>>> bset = {12, 23, 33}
>>> aset | bset
{33, 22, 23, 11, 12}
```

Intersection of two sets:

```
>>> aset & bset
{33}
```

Difference:

```
>>> aset - bset
{11, 22}
```

Symmetric difference:

```
>>> aset ^ bset
{11, 12, 22, 23}
```

Dictionaries

Dictionaries (I)

- Store ***pairs*** of entries called ***items***
  { 'Belgium' : 'Brussels', 'Spain' : 'Madrid'}
- Each pair of entries contains
  - A ***key***
  - ***A value***
- Key and values are separated by a colon
- Paris of entries are separated by commas
- Dictionary is enclosed within curly braces

Usage

- Keys must be ***unique*** within a dictionary
- No ***duplicates***
- If we have
  age = {'Anna' : 25, 'Ben' :28}
  then
  age['Anna'] is **25**
  and
  age[Ben'] is **28**

Dictionaries are mutable

```
>>> age = {'Anna' : 25, 'Ben' :28}
>>> saved = age
>>> age['Ben'] = 29
>>> age
{'Anna': 25, 'Ben': 29}
>>> saved
{'Anna': 25, 'Ben': 29}
```

Keys must be unique

```
>>> age = {'Anna' : 25, 'Ben' : 28, 'Anna' : 26}
>>> age
{'Anna': 26, 'Ben': 28}
```

Displaying contents

```
>>> age = {'Anna' : 25, 'Ben' : 'twenty-eight'}
>>> age.items()
dict_items([('Anna', 25), ('Ben', 'twenty-eight')])
>>> age.keys()
dict_keys(['Anna', 'Ben'])
>>> age.values()
dict_values([25, 'twenty-eight'])
```

Updating directories

```
>>> age = {'Anna' : 25, 'Ben' : 28}
>>> age.update({'Ben' : 29})
>>> age
{'Anna': 25, 'Ben': 29}
>>> age.update({'Anna' : 28})
>>> age
{'Anna': 28, 'Ben': 29}
```

Returning a value

```
>>> age = {'Anna' : 25, 'Ben' :28}
>>> age.get('Ben')
28
>>> age['Ben']
28
```

Removing a specific item

```
>>> age = {'Anna' : 25, 'Ben' : 'twenty-eight'}
>>> age
{'Anna': 25, 'Ben': 'twenty-eight'}
>>> age.pop('Anna')
25
>>> age
{'Ben': 'twenty-eight'}
>>> age.pop('Ben')
'twenty-eight'
>>> age
{}
```

Remove a random item

```
>>> age = {'Anna' : 25, 'Ben' : 28, 'Charlie': 31}
>>> age.popitem()
('Charlie', 31)
>>> age
{'Anna': 25, 'Ben': 28}
>>> age.popitem()
('Ben', 28)
>>> age
{'Anna': 25}
```

Summary

- Strings, lists, tuples, sets and dictionaries all deal with aggregates
- Two big differences
  - ***Lists*** and ***dictionaries*** are ***mutable***
    - Unlike strings, tuples and sets
  - ***Strings***, ***lists*** and ***tuples*** are ***ordered***
    - Unlike sets and dictionaries

Mutable aggregates

- Can modify individual items
  - **x = [11, 22, 33]**
    x[0] = 44
    will work
- Cannot save current value
  - **x = [11,22, 33]**
    y = x
    will **not** work

Immutable aggregates

- Cannot modify individual items
  - **s = 'hello!'**
    s[0] = 'H'
    is an **ERROR**
- Can save current value
  - **s= 'hello!'**
    t = s
    will work

Ordered aggregates

- Entities in the collection can be accessed through a **numerical index**
  - **s= 'Hello!'**
    s[0]
  - **x = ['Alice', 'Bob', 'Carol']**
    x[-1]
  - **t = (11, 22)**
    t[1]

Other aggregates

- Cannot index sets
  - **myset = {'Apples', 'Bananas', 'Oranges'}**
  - **myset[0] is WRONG**
- Can only index dictionaries **through their keys**
  - **age = {'Ben': 29, 'Charlie': 23, 'Anna': 26}**
    age['Anna'] works
    age[0] is **WRONG**