Python Collections - LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

Python Collections - LIST

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Empty List:

```
my_list = []
```

Types of list

```
my_list = [10, ‘Howdy’, [‘Strawberry’, ‘Peach’]] #different type list
```

Accesses value in list:

```
#!/usr/bin/python
list1 = ['physics', 'chemistry', 1997, 2000]; list2 = [1, 2, 3, 4, 5, 6, 7 ];
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

Python Collections - LIST

updating list:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Delete list element:

```
del thislist[1]
print(thislist)
```

List operation:

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Python Collections - LIST

List methodes:

- my_list.append(obj) #Appends object obj to list
- my_list.count(obj) #Returns count of how many times obj occurs in list
- my_list.extend(seq) #Appends the contents of seq to list
- my_list.insert(index, obj) #Inserts object obj into list at offset index
- my_list.pop(obj=list[-1]) #Removes and returns last obj from list
- my_list.remove(obj) #Removes object obj from list
- my_list.reverse() #Reverses objects of list in place
- my_list.sort() #Sorts objects of list

len(my_list) #return length of list

The list() Constructor thislist =

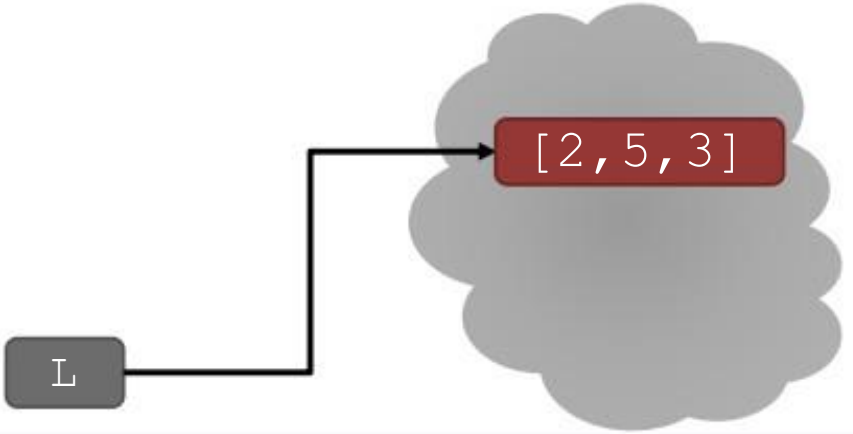
list(("apple", "banana", "cherry")) # note the double round-brackets

CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0
  for i in range(len(L)):
      total += L[i]
  print total
```

```
total = 0
  for i in L:
      total += i
  print total
```

like strings, can iterate over list elements directly

- notice
  - list elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

```
L = [2,1,3]
L.append(5)
```

- L is now `[2,1,3,5]`

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by `object_name.do_something()`
  - will learn more about these later

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
```

- `L3` is `[2,1,3,4,5,6]`
  `L1, L2` unchanged

```
L1.extend([0,6])
```

- mutated `L1` to `[2,1,3,0,6]`

OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del(L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
  - looks for the element and removes it
  - if element occurs multiple times, removes first occurrence
  - if element not in list, gives an error

all these operations mutate the list

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2)  · mutates L = [1,3,6,3,7,0]
L.remove(3)  · mutates L = [1,6,3,7,0]
del(L[1])    · mutates L = [1,3,7,0]
L.pop()      · returns 0 and mutates L = [1,3,7]
```

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

`s = "I<3 cs"`	`s` is a string
`list(s)`	returns `['I','<','3',' ','c','s']`
`s.split('<')`	returns `['I', '3 cs']`
`L = ['a','b','c']`	`L` is a list
`''.join(L)`	returns `"abc"`
`'_'.join(L)`	returns `"a_b_c"`

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!

https://docs.python.org/3/tutorial/datastructures.html

```
L=[9,6,0,3]
```

`sorted(L)`	• returns sorted list, does **not mutate** `L`
`L.sort()`	• **mutates** `L=[0,3,6,9]`
`L.reverse()`	• **mutates** `L=[9,6,3,0]`

LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

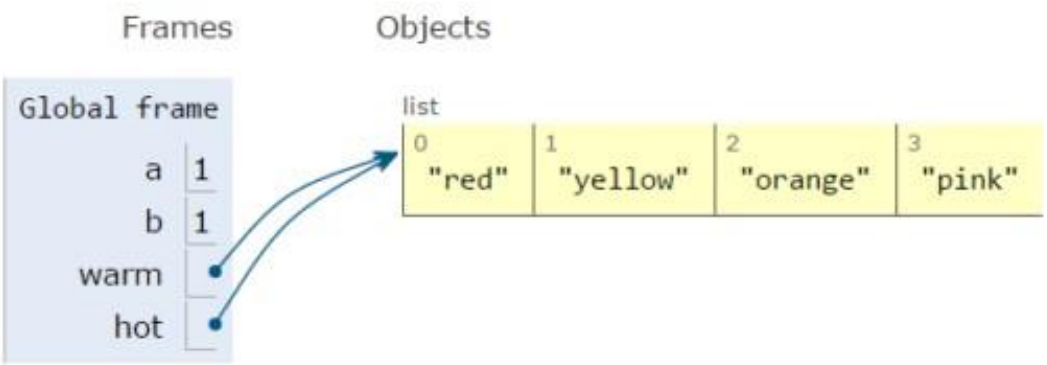
ALIASES

- `hot` is an **alias** for `warm` - changing one changes the other!
- `append()` has a side effect

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
hot.append('pink')
print(hot)
print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

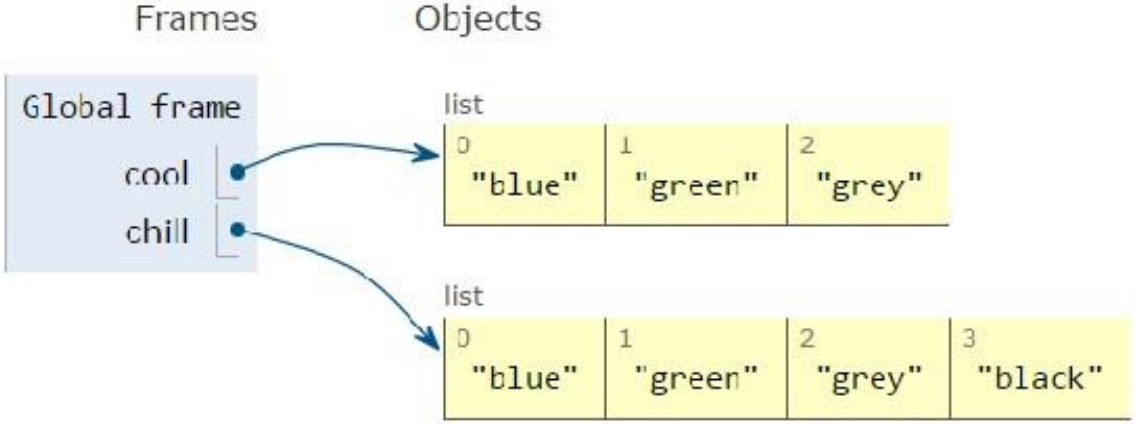


CLONING A LIST

- create a new list and **copy every element** using `chill = cool[:]`

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```



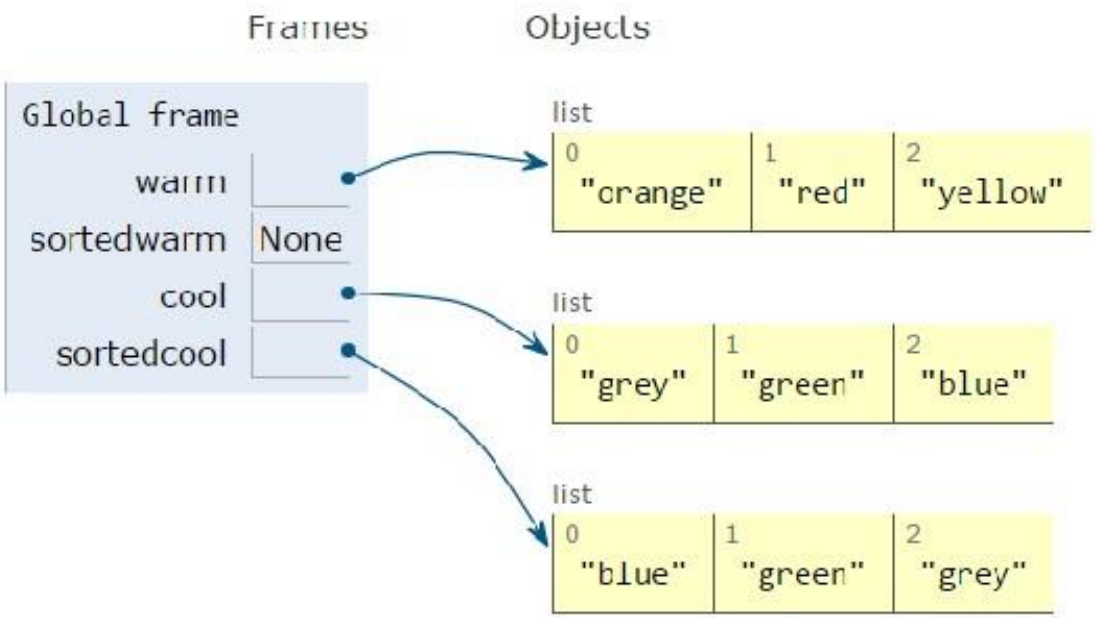
SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```

```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

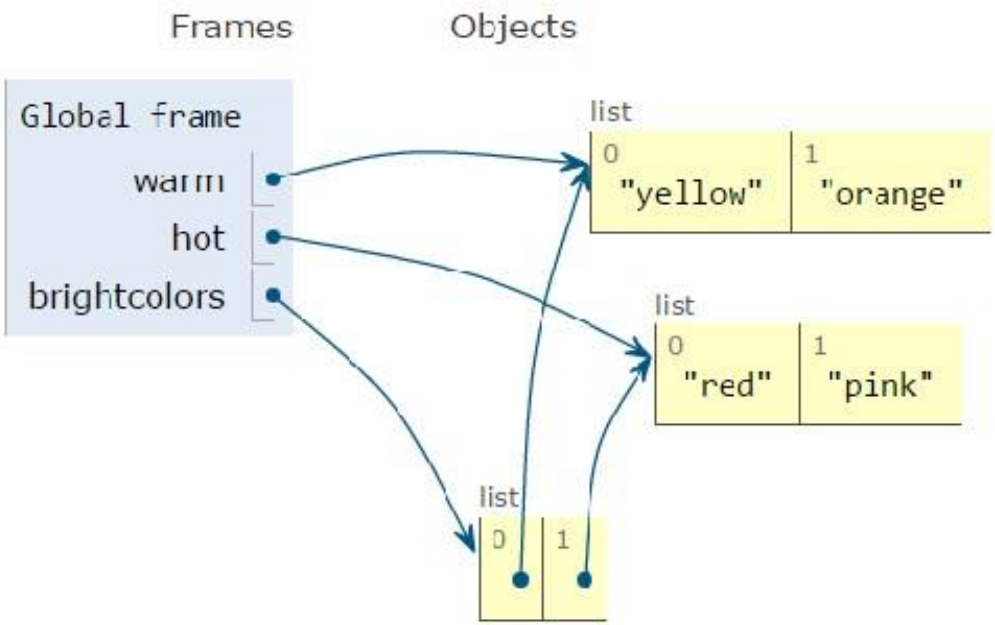


LISTS OF LISTS OF LISTS OF.*

- can have **nested** lists
- side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



MUTATION AND ITERATION
Try this in Python Tutor!

- **avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

clone list first, note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

```
te = ()
```

empty tuple

```
t = (2,"mit",3)
```

`t[0]` • evaluates to 2

`(2,"mit",3) + (5,6)` • evaluates to `(2,"mit",3,5,6)`

`t[1:2]` • slice tuple, evaluates to `("mit",)`

`t[1:3]` • slice tuple, evaluates to `("mit",3)`

`len(t)` • evaluates to 3

`t[1] = 4` • gives error, can't modify object

extra comma means a tuple with one element

TUPLES

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Return the item in position 1:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

TUPLES

Change Tuple Values:

```
#!/usr/bin/python
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
# Following action is not valid for tuples
# tup1[0] = 100;
```

Delete Tuple Elements:

```
#!/usr/bin/python
 tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del (tup)
print ("After deleting tup : “)
print (tup)
```

TUPLES

- conveniently used to **swap** variable values

```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
```

integer division

TUPLES

Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely:

- my_tuple = () #empty tuple

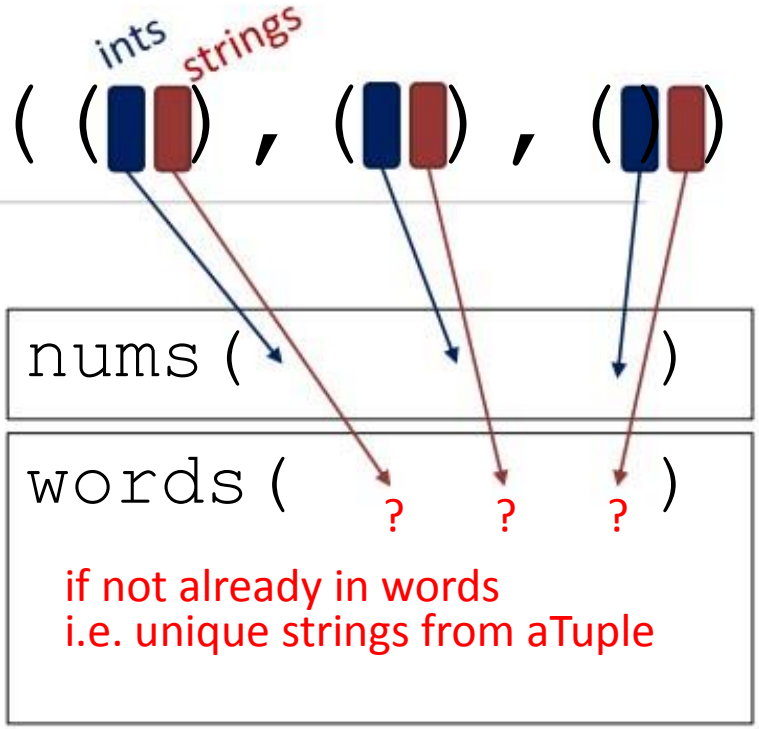
The tuple() Constructor:

thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)

Tuple Object Methods:

- cmp(tuple_1, tuple_2) #compares two tuples elements, return 0 if true -1 if false.
- len(tuple_1) # returns the length of a given tuple.
- max(tuple_1) #returns the max value element of a given tuple.
- min(tuple_1) #returns the max value element of a given tuple.
- tuple(list_1) #converts a given list into a tuple.

MANIPULATING TUPLES



- can **iterate** over tuples

```
def get_data(aTuple):
    nums = ()
    words = ()
    for t in aTuple:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words +
            (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

empty tuple

singleton tuple

SET

A set is a collection which is unordered and unindexed.
In Python sets are written with curly brackets.:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Access Items

-By Looping

```
for x in thisset:
  print(x)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

SET

Add Items

To add one item to a set use the add() method
To add more than one item to a set use the update() method

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

thisset.update(["orange", "mango", "grapes"])
```

Get the Length of a Set

Len() as list

SET

Remove Item

To remove an item in a set, use the remove(), or the discard() method

thisset.remove("banana")

Note: discart will not raise error if you try remove item not exist but remove will do.

Some method for set

pop() #as list

clear() #clear set as list

del() #as list

set() #constructor

DICTIONARIES

A BETTER AND CLEANER WAY - A DICTIONARY

§ nice to **index item of interest directly** (not always int)

§ nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dic0onary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

A PYTHON DICTIONARY

§ store pairs of data
- key
- value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

custom index by label

element

empty dictionary

```
my_dict = {}

grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'
```

key1 val1 key2 val2 key3 val3 key4 val

DICTIONARY LOOKUP

§ similar to indexing into a list

§ **looks up** the **key**

§ **returns** the **value** associated with the key

§ if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A
```

`grades['John']` à evaluates to `'A+'`

`grades['Sylvan']` à gives a `KeyError`

DICTIONARIES

Also known as “Key-Value binding” Or Hashing

my_dic = {‘key’ : ‘value’}

*The Value can be anything. List, tuple, string, int, or even another dictionary.

*The key shouldn’t be changed as it is our reference to the value. So we can use tuples as a dictionary key as long as tuples are immutable.

Empty Dict:

thisdict = {}

DICTIONARIES

Examples:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Accessing Items

```
x = thisdict["model"]
x = thisdict.get("model") #the get() method taken key as arg and retunred value
```

Change Values

```
thisdict["year"] = 2018
```

DICTIONARIES

Dictionary Object Methods:

dict.clear() #remove all elements of dictionary dict.

dict.copy() #returns a copy of dictionary dict.

dict.items() #returns List of dict (keys, values) Tuple pairs.

dict.keys() #returns a List of dict keys.

dict.values() #return a List of dict values.

dict_1.update(dict_2) #add dict_2 elements (key, value) to dict_1

dict.popitem() #method removes the last inserted item (in versions before 3.7, a

#random item is removed instead)

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'
```

§ **add** an entry

```
grades['Sylvan'] = 'A'
```

§ **test** if key in dicSonary

```
'John' in grades
'Daniel' in grades
```

à returns `True`
à returns `False`

§ **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'
```

§ get an **iterable that acts like a tuple of all keys**

no guaranteed order

`grades.keys()` à returns `['Denise','Katy','John','Ana'`

§ get an **iterable that acts like a tuple of all values**

`grades.values()` à returns `['A', 'A', 'A+', 'B']`

no guaranteed order

DICTIONARY KEYS and VALUES

§ values
- any type (**immutable and mutable**)
- can be **duplicates**
- dicSonary values can be lists, even other dicSonaries!

§ keys
- must be **unique**
- **immutable** type (`int, float, string, tuple,bool`)
  - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
- careful with `float` type as a key

§ **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

`list` vs	`dict`
§ **ordered** sequence of elements	§ **matches** “keys” to “values”
§ look up elements by an integer index	§ look up one item by another item
§ indices have an **order**	§ **no order** is guaranteed
§ index is an **integer**	§ key can be any **immutable** type

ARRAYS

(One Type List)

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

To use arrays we need to import it from array module.

```
from array import array
```

Array structure:

```
my_arr = array('data_type_code', initial_values)
```

Then my_arr.append(values) to expand it.

Data Type codes: B, i for Integers.....there are more.