

Data Collections (Sequences)

It's complicated

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Lists and Tuples

- Belongs to a type of data structure called arrays
- Intrinsic ordering
 - Easy to retrieve data from the array
- Lists
 - Mutable
 - Dynamic Arrays
- Tuples
 - Immutable
 - Static Arrays

Lists

- Ordered sequence of data types

- Homogeneous sequence

- Sequence of integers
 - Sequence of floats
 - Sequence of strings
 - Sequence of lists
 - Sequence of functions, etc.

- Heterogeneous sequence

- mix of integers, floats, strings, etc.

```
>>> empty_list = []
>>> empty_list
[]
>>> int_list = [1,2,3,4]
>>> int_list
[1, 2, 3, 4]
>>> float_list = [1.0,2.0,3.0,4.0]
>>> float_list
[1.0, 2.0, 3.0, 4.0]
>>> string_list = ['Hello!', 'Welcome', 'to', 'IT5001']
>>> string_list
['Hello!', 'Welcome', 'to', 'IT5001']
>>> heterogeneous_list = [1,2.0,'Hello', 3+4j]
>>> heterogeneous_list
[1, 2.0, 'Hello', (3+4j)]
```

- Defined using square brackets - []

Lists are Referential Arrays

Python 3.6
(known limitations)

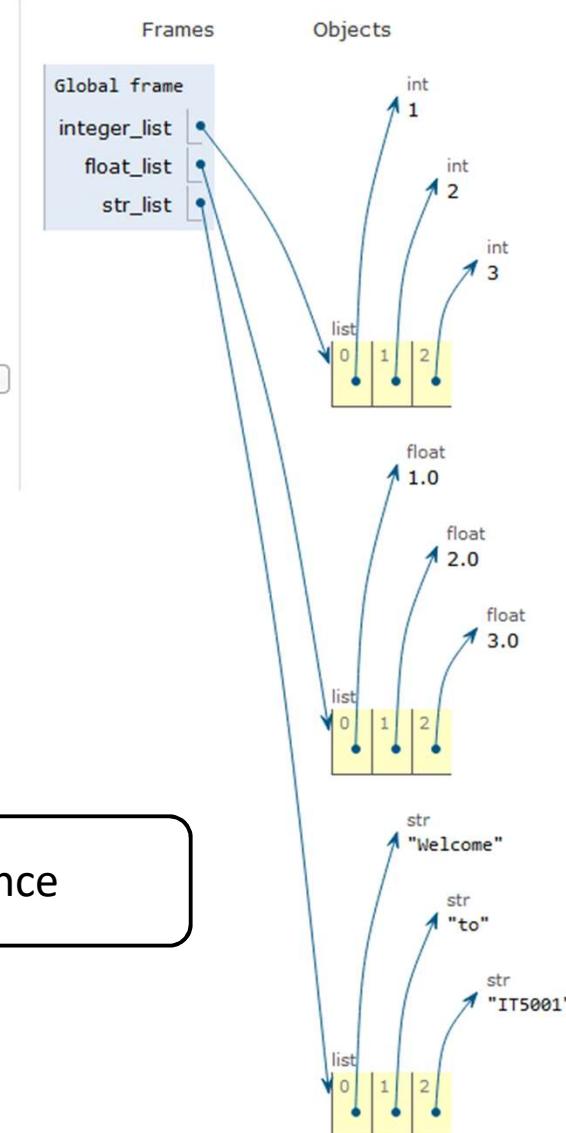
```
1 integer_list = [1,2,3]
2
3 float_list = [1.0, 2.0, 3.0]
4
5 str_list = ['Welcome', 'to', 'IT5001']
```

[Edit this code](#)

Just executed
o execute

<< First < Prev Next > Last >>

Done running (3 steps)



Lists store addresses of its items in sequence

All Indexed Sequences can...

a[i]	return i-th element of a
a[i:j]	returns elements i up to j-1
len(a)	returns numbers of elements in sequence
min(a)	returns smallest value in sequence
max(a)	returns largest value in sequence
x in a	returns True if x is a part of a
a + b	concatenates a and b
n * a	creates n copies of sequence a

```
>>> int_list = [1,2,3,4]
>>> print(f'Element at index = 1 is {int_list[1]}')
Element at index = 1 is 2
>>> print(f'Elements at index = [1,3) are {int_list[1:3]}')
Elements at index = [1,3) are [2, 3]
>>> print(f'Number of Elements = {len(int_list)}')
Number of Elements = 4
>>> print(f'Smallest Element in the List = {min(int_list)}')
Smallest Element in the List = 1
>>> print(f'Largest Element in the List = {max(int_list)}')
Largest Element in the List = 4
>>> print(f'Is element 2 in the list: {2 in int_list}')

Is element 2 in the list: True
>>> another_int_list = [5,6,7,8]
>>> print(f'Concatenated List: {int_list+another_int_list}')
Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8]
>>> print(f'Repeated list: {int_list*3}')
Repeated list: [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Lists are mutable

- Elements can be replaced
- Elements can be added
- Elements can be removed
 - A specific element
 - If element occurs multiple times, removes first occurrence
 - Element at a specific location (index)
 - From the end of the list
- Elements can be sorted
 - `sort()`
 - `Sorted()`
- Elements can be reversed

Lists are Dynamic-Size Arrays

Elements can be added (appended) to the list

```
>>> id(integer_list)  
1421979130760  
>>> integer_list  
[2, 5, 3, 4]  
>>> integer_list.append(9)  
>>> integer_list  
[2, 5, 3, 4, 9]  
>>> id(integer_list)  
1421979130760
```

Elements can be removed from the list

```
>>> integer_list = [2,5,3,4,9]  
>>> id(integer_list)  
2378756596296  
>>> integer_list.remove(9)  
>>> integer_list  
[2, 5, 3, 4]  
>>> id(integer_list)  
2378756596296  
>>> my_list = [1,2,3,4,6,3]  
>>> my_list.remove(3)  
>>> my_list  
[1, 2, 4, 6, 3]
```

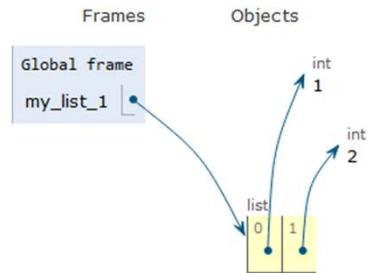
Append Vs Concatenation

```
Python 3.6  
(known limitations)
```

```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

[Edit this code](#)

at executed
execute



1

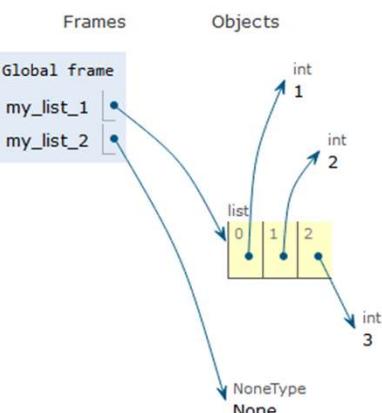
```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

[Edit this code](#)

at executed
execute

[**<< First**](#) [**< Prev**](#) [**Next >**](#) [**Last >>**](#)

Step 3 of 3



2

Append Vs Concatenation

Python 3.6
([known limitations](#))

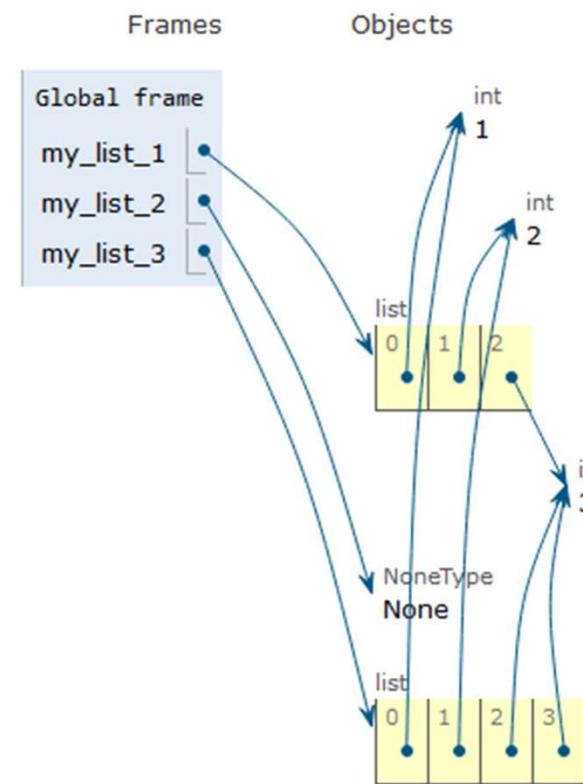
```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

[Edit this code](#)

Just executed
to execute

<< First < Prev Next > Last >>

Done running (3 steps)



3

Append Vs Concatenation

```
>>> my_list_1 = [1,2,3,4]
>>> my_list_2 = [5,6,7,8]
>>> my_list_1.append(my_list_2)
>>> my_list_1
[1, 2, 3, 4, [5, 6, 7, 8]]
>>> my_list_1 = my_list_1 + my_list_2
>>> my_list_1
[1, 2, 3, 4, [5, 6, 7, 8], 5, 6, 7, 8]
```

Strings to Lists and Vice-Versa

Python 3.6
[\(known limitations\)](#)

```
1 my_string = "IT5001"
2
3 my_list = list(my_string)
```

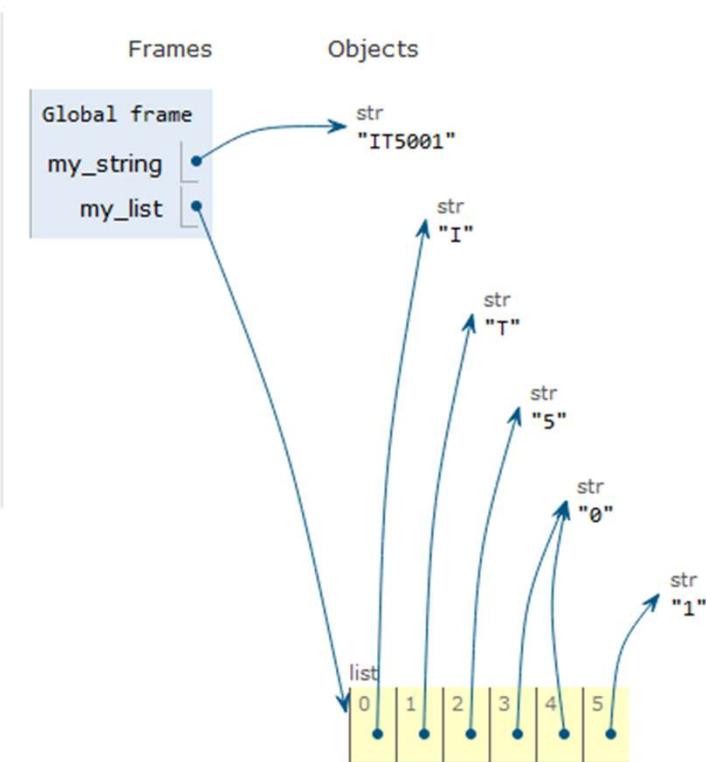
[Edit this code](#)

Just executed
ine to execute

[`<< First`](#) [`< Prev`](#) [`Next >`](#) [`Last >>`](#)

Done running (2 steps)

[: visualization](#)



Strings to Lists and Vice-Versa

Python 3.6
([known limitations](#))

```
1 my_list = ['a', 'b', 'c', 'd']
2 my_string = str(my_list)
→ 3 a = my_string[0]
```

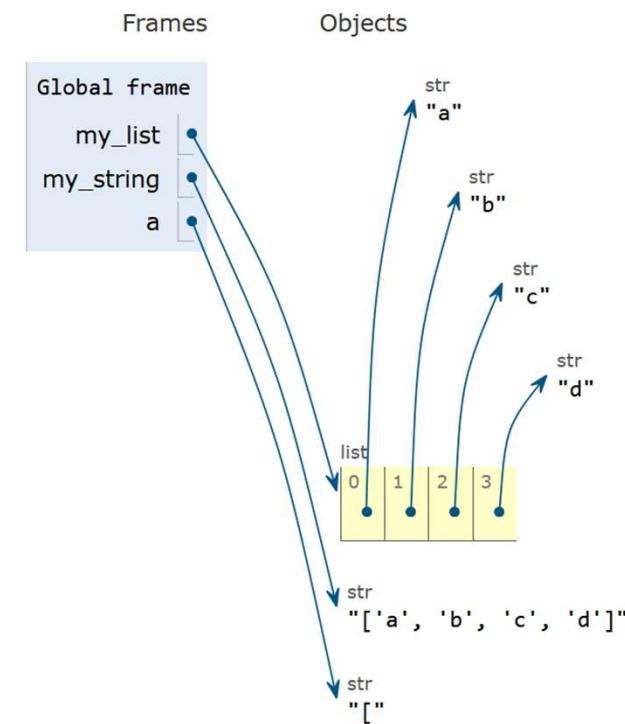
[Edit this code](#)

Line status: green arrow line that just executed
red arrow next line to execute

<< First < Prev Next > Last >>

Done running (3 steps)

[Customize visualization](#)



Aliasing vs Cloning

Aliasing

```
Python 3.6  
(known limitations)
```

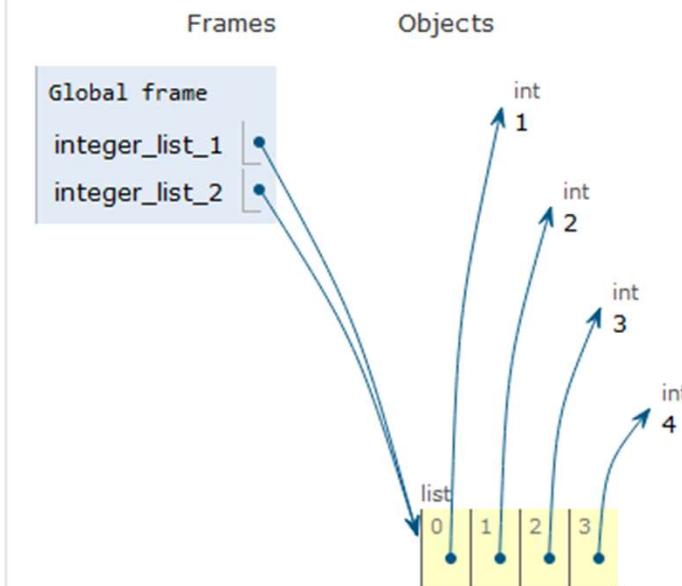
```
1 integer_list_1= [1,2,3,4]
2
3
4 integer_list_2 = integer_list_1
5
6
7 integer_list_3 = integer_list_1[:]
```

[Edit this code](#)

it just executed
ie to execute

[**<< First**](#) [**< Prev**](#) [**Next >**](#) [**Last >>**](#)

Step 3 of 3



Aliasing vs Cloning

Python 3.6
([known limitations](#))

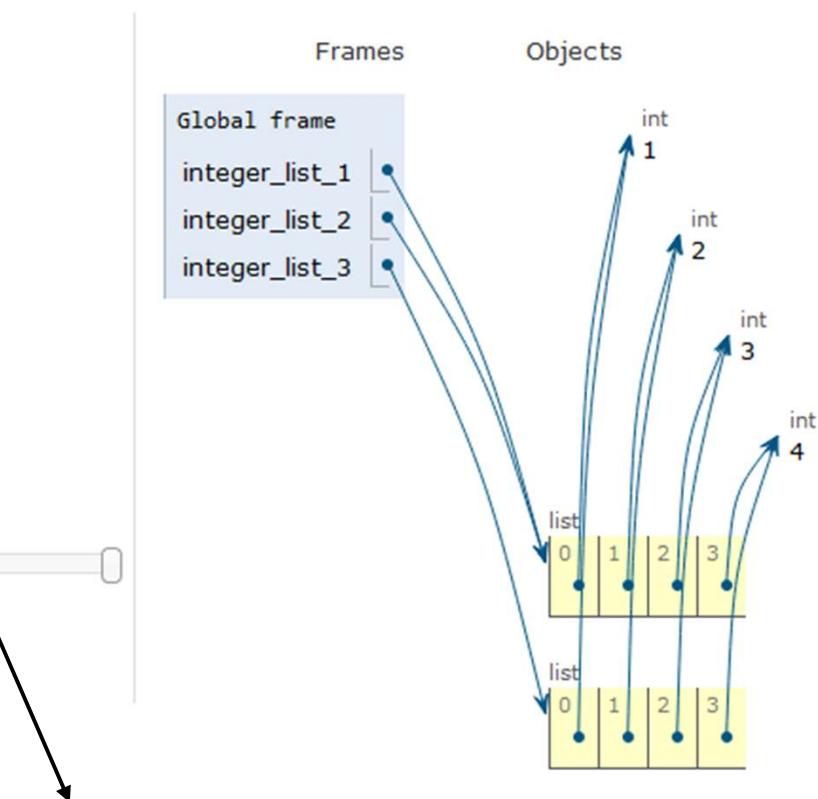
```
1 integer_list_1= [1,2,3,4]
2
3
4 integer_list_2 = integer_list_1
5
6
7 integer_list_3 = integer_list_1[:]
```

[Edit this code](#)

just executed
to execute

<< First < Prev Next > Last >>

Done running (3 steps)



Cloning

Sort vs Sorted

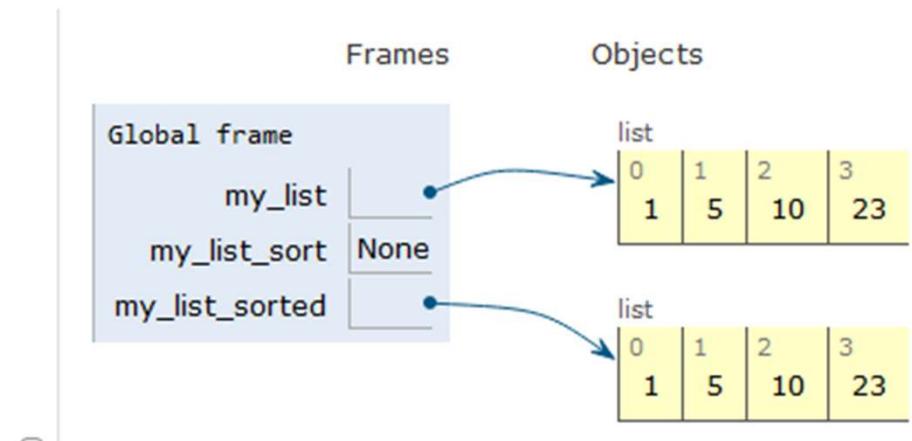
- `sort()` method mutates the list
- `sorted()` method creates a new sorted list without mutating the original list

Python 3.6
([known limitations](#))

```
1 my_list = [10, 1, 23, 5]
2 my_list_sort = my_list.sort()
3 my_list_sorted = sorted(my_list)
```

[Edit this code](#)

executed
execute



Reverse

- `reverse()` method mutates the list

Python 3.6
[\(known limitations\)](#)

```
1 integer_list = [1,2,3,4]
2
3 integer_list.reverse()
```

[Edit this code](#)



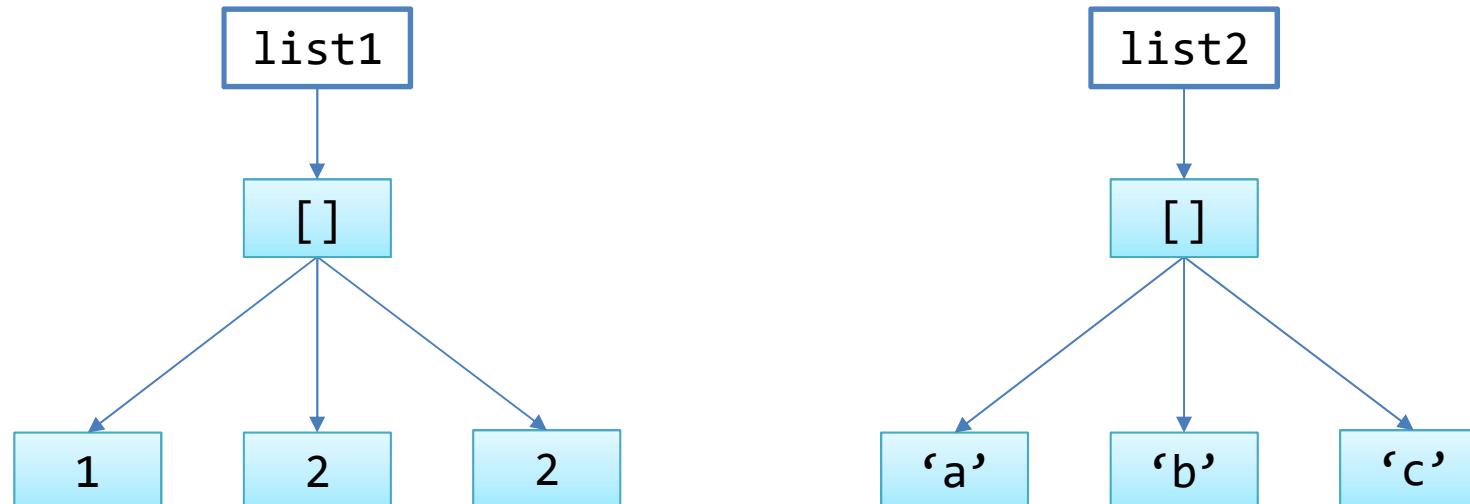
Lists of Anything

- A list of
 - Lists?

```
>>> list1 = [1,2,3]
>>> list2 = ['a','b','c']
>>> list3 = [list1,list2]
>>> list3
[[1, 2, 3], ['a', 'b', 'c']]
>>> list4 = [True,list3,list1]
>>> list4
[True, [[1, 2, 3], ['a', 'b', 'c']], [1, 2, 3]]
```

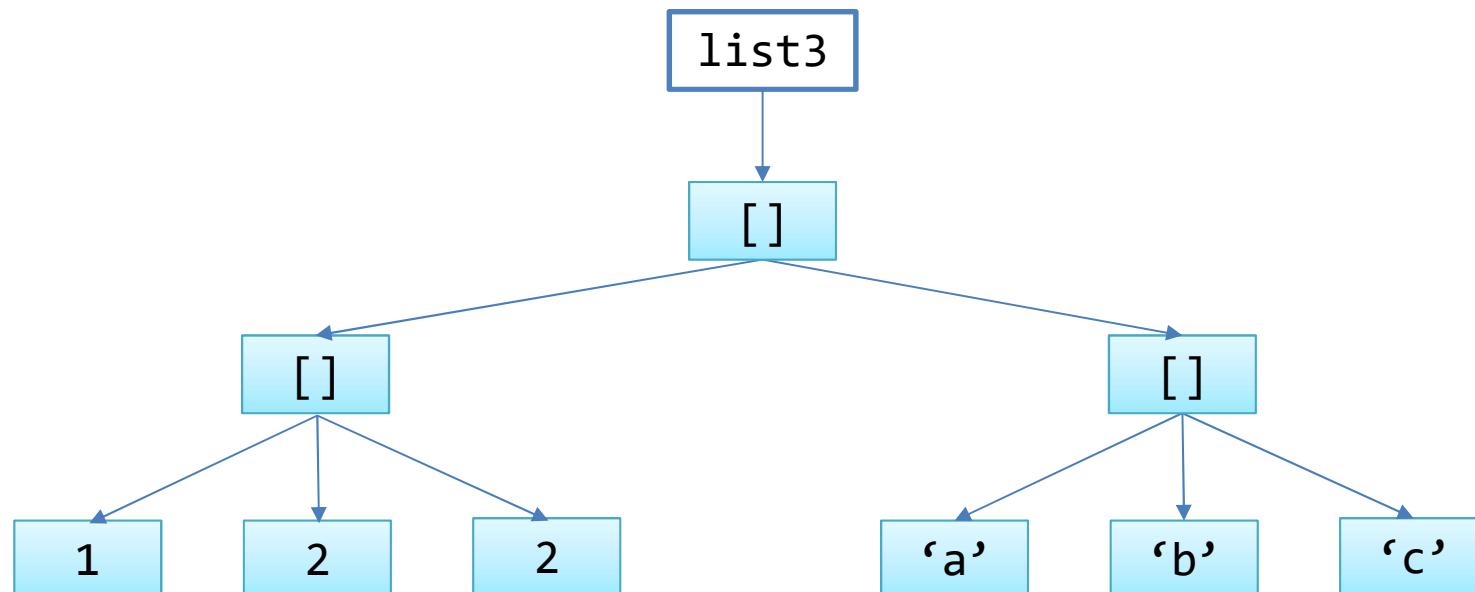
Block Diagram

```
>>> list1 = [1,2,3]  
>>> list2 = ['a','b','c']
```



Block Diagram

```
>>> list3 = [list1,list2]  
>>> list3  
[[1, 2, 3], ['a', 'b', 'c']]
```



Lists are Iterable

due to this method

```
>>> my_list = [1,2,3,4]
>>> dir(my_list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__has__',
 '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
```

```
>>> for e in my_list:
    print(e)
```

1
2
3
4

For loop

```
>>> for i in range(0,5):  
    print (i)
```

0
1
2
3
4

```
>>> for i in [0,1,2,3,4]:  
    print(i)
```

0
1
2
3
4

For Loop

```
my_list = [1,2,3,4]
```

```
for i,j in enumerate(my_list):  
    print(f'Element at index {i} is {j}')
```

Output:

```
Element at index 0 is 1  
Element at index 1 is 2  
Element at index 2 is 3  
Element at index 3 is 4
```

Never do this

```
myList = [1, 2, 3, 4]  
  
for ele in myList:  
    myList.remove(ele)
```

Why?

Mutation and Iteration

- Avoid mutating a list while iterating over the list

```
myList = [1, 2, 3, 4]  
  
for ele in myList:  
    myList.remove(ele)
```

- *next()* method uses an index to retrieve the elements from *myList*
- *remove()* method mutates the *myList*, but the index in *next()* method will not be updated

Example: Find Max in A List of No.

```
list1 = [2,101,3,1,6,33,22,4,99,123,55]
```

```
def findMax(lst):
    maxsofar = lst[0]
    for i in lst:
        if i > maxsofar:
            maxsofar = i
    return maxsofar

>>> print(findMax(list1))
123
```

- Is there any potential problem?

Example: Find all Even Numbers

```
def findAllEvenNo(lst):
    output = []
    for i in lst:
        if i % 2 == 0:
            output.append(i)
    return output

>>> print(findAllEvenNo(list1))
[2, 6, 22, 4]
```

List Comprehensions

- Provides a concise way to apply an operation to the items in iterable object and store the result as a list
- Syntax:
 - `[expr for elem in iterable if test]`
- Returns an iterable

List Comprehension

- Todo:
 - create a list:

```
a_list = [1,2,3,4,5,6,..... , 100]
```

- You can

```
>>> a_list = []
>>> for i in range(1,101):
    a_list.append(i)
```

```
>>> a_list
[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, 100]
```

List Comprehension

- Or

The item really in the list

every i between 1 and
101 (exclusive)

```
>>> b_list = [ i for i in range(1,101) ]  
>>> b_list  
[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, 100  
]
```

$$b = \{i | i \in [1, 101)\}$$

Compare to
ordinary math
equation

List Comprehension

- How do I produce a list of first 10 squared numbers?

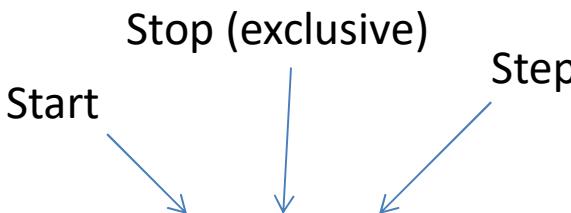
```
>>> d_list = [i*i for i in range(1,11)]  
>>> d_list  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

$$b = \{i^2 | i \in [1,101)\}$$

Compare to
ordinary math
equation

List Comprehension

- How do I produce a list of odd numbers less than 100
 - Like string slicing



```
>>> c_list = [i for i in range(1,101,2)]
>>> c_list
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29,
31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57,
59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85,
87, 89, 91, 93, 95, 97, 99]
```

List Comprehension

- How do I produce a list of **even** numbers less than 100
 - Similar to the previous one but start with 2
 - Or

```
>>> c2_list = [i for i in range(1,101) if i not in c_list]
>>> c2_list
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
```

Advance: Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers <50

```
>>> for i in range(2,8):  
    print([j for j in range(i*2, 50, i)])
```

i is from 2 to 7
($7 = \sqrt{50}$)

get all the multiples of i
from $2*i$ to 49

```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,  
34, 36, 38, 40, 42, 44, 46, 48]  
[6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]  
[8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48]  
[10, 15, 20, 25, 30, 35, 40, 45]  
[12, 18, 24, 30, 36, 42, 48]  
[14, 21, 28, 35, 42, 49]
```

Advance: Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers <50

```
>>> nonprime =[j for i in range(2,8) for j in range(i*2, 50, i)]  
>>> nonprime  
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,  
, 38, 40, 42, 44, 46, 48, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33,  
36, 39, 42, 45, 48, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 1  
0, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 2  
1, 28, 35, 42, 49]
```

i is from 2 to 7

get all the multiples of i from $2 \cdot i$ to 49

i = 2

i = 3

i = 4

Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers <50
- Prime numbers are the numbers NOT in the list above

```
>>> nonprime =[j for i in range(2,8) for j in range(i*2, 50, i)]
>>> prime = [x for x in range(1,50) if x not in nonprime]
>>> prime
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Generator Expressions

- Provides a generator that can be used to iterate over without explicitly generating the list of items
- Syntax:
 - *(expr for elem in iterable if test)*
- Returns an iterator
- Requires less memory than list
 - Check!

Generator Expressions

```
num = 4
# square each term using list comprehension
my_square_list = [x**2 for x in range(num)]
my_square_generator = (x**2 for x in range(num))

for k in my_square_list:
    print(k)

for k in my_square_generator:
    print(k)
```

Output:

```
0  
1  
4  
9  
0  
1  
4  
9
```

Which is better?

Generator Expressions

```
num = 10**4
# square each term using list comprehension
my_square_list = [x**2 for x in range(num)]
my_square_generator = (x**2 for x in range(num))

import sys
print(f'Size of my_square_list {sys.getsizeof(my_square_list)}')
print(f'Size of my_square_generator {sys.getsizeof(my_square_generator)}')
```

Output:

```
Size of my_square_list 87624
Size of my_square_generator 120
```

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - **Tuples**
- Non-indexed collection:
 - Sets
 - Dictionary

Tuples

- A static and an immutable array/list

- Syntax:

- int_tuple = (1,2,3)
- float_tuple = (1.0,2.0,3.0)
- str_tuple = ('hi','IT5001')
- mixed_tuple = (1,1.0,'IT5001')

Task	Syntax
Return i-th element	a[i]
Return elements from i to j-1	a[i:j]
Return number of elements	len(a)
Return smallest value in sequence	min(a)
Return largest value in sequence	max(a)
Returns if an element is part of sequence	x in a
Concatenates two sequences	a + b
Creates n copies of a sequence	a * n

Tuples: Example 1

Python 3.6
[\(known limitations\)](#)

```
1 my_tuple_1 = (1,2,3,4)
2
3 my_tuple_2 = my_tuple_1
```

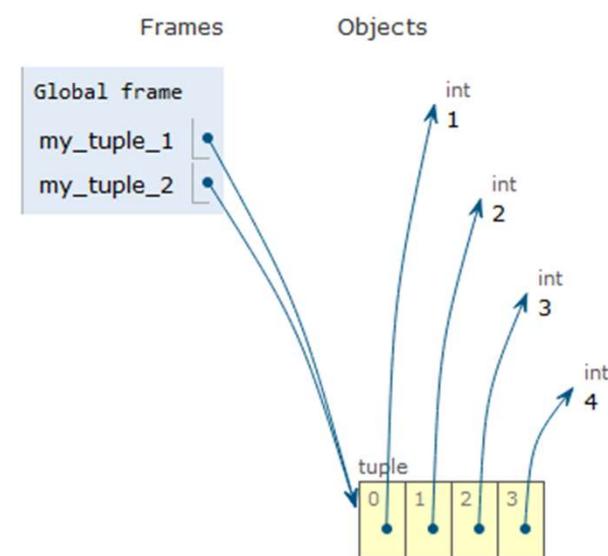
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Done running (2 steps)

[Customize visualization](#)



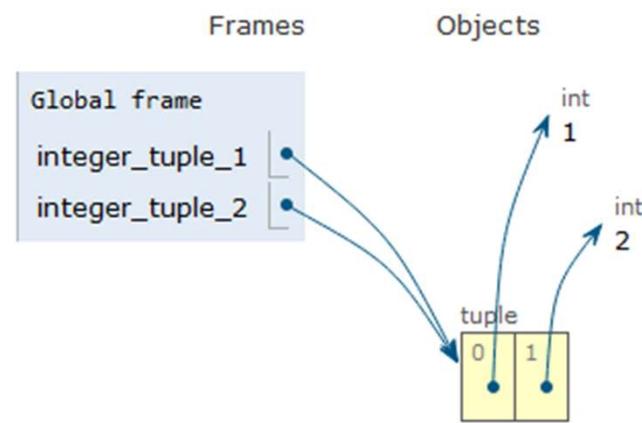
Tuples: Example 2

Python 3.6
([known limitations](#))

```
1 integer_tuple_1= (1,2)
2
3 integer_tuple_2 = integer_tuple_1
4
5 integer_tuple_1 + (3,4)
6
7 integer_tuple_1 = integer_tuple_1 + (3,4)
```

[Edit this code](#)

that just executed
: line to execute



Immutable:

creates a new tuple – but not assigned

Tuples: Example 2

Python 3.6
(known limitations)

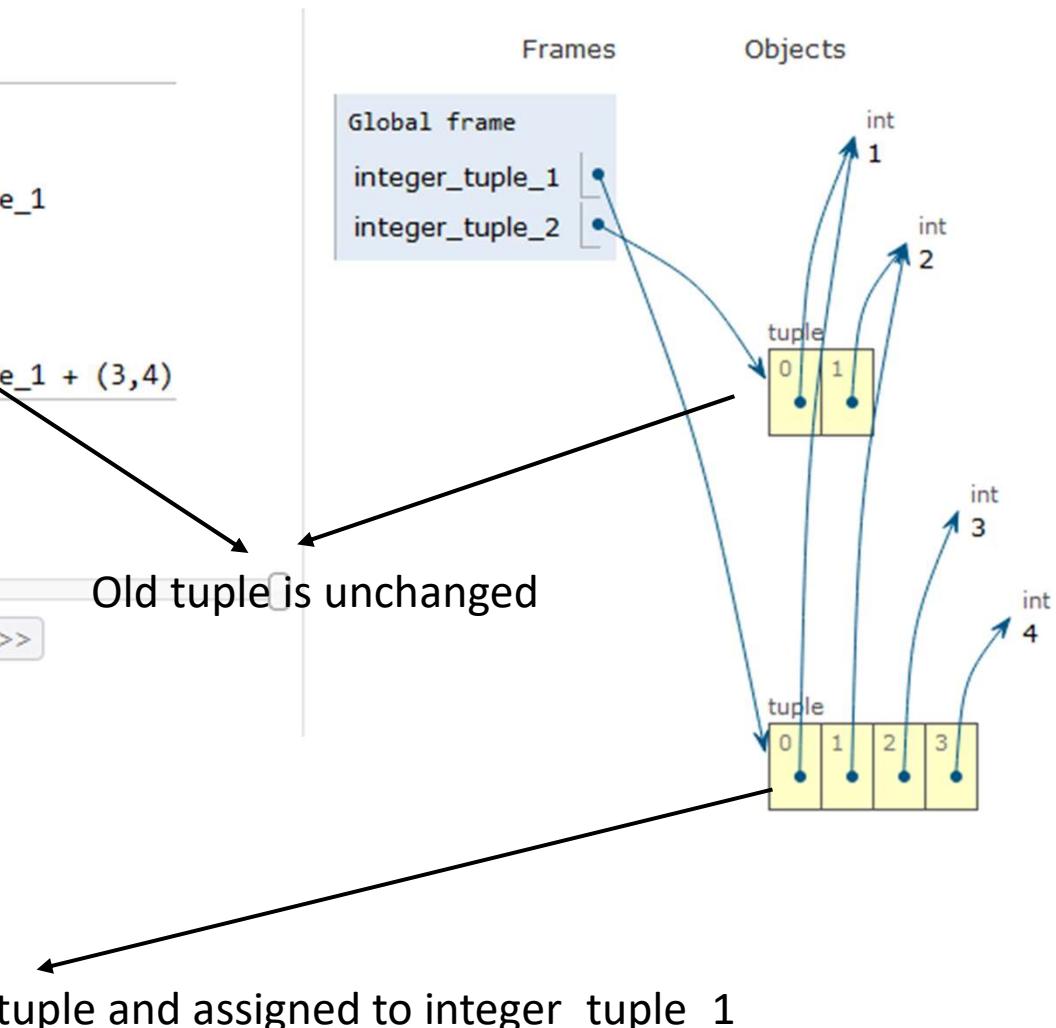
```
1 integer_tuple_1= (1,2)
2
3 integer_tuple_2 = integer_tuple_1
4
5 integer_tuple_1 + (3,4)
6
7 integer_tuple_1 = integer_tuple_1 + (3,4)
```

[Edit this code](#)

at just executed
ine to execute

<< First < Prev Next > >>
Done running (4 steps)

[visualization](#)



Lists and Tuples

- Similarities:
 - List and Tuple are
 - Indexed
 - Iterable
 - Both can store heterogeneous data types
- Differences:
 - List is mutable
 - Tuple is immutable

Tuple

- A Tuple is basically a list but
 - CANNOT be modified

```
>>> a_tuple = (12, 13, 'dog') ← Tuples use '(' and ')'  
>>> a_tuple[1] ← Lists use '[' and ']'  
13  
>>> a_tuple[1] = 9  
Traceback (most recent call last):  
  File "<pyshell#130>", line 1, in <module>  
    a_tuple[1] = 9  
TypeError: 'tuple' object does not support item assignment  
>>> a_tuple.append(1)  
Traceback (most recent call last):  
  File "<pyshell#131>", line 1, in <module>  
    a_tuple.append(1)  
AttributeError: 'tuple' object has no attribute 'append'  
>>>
```

Tuple

- A Tuple is basically a list but
 - CANNOT be modified

```
>>> t1 = (1,2,3)
>>> t1.append(3)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t1.append(3)
AttributeError: 'tuple' object has no attribute 'append'
>>> t1.remove(1)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    t1.remove(1)
AttributeError: 'tuple' object has no attribute 'remove'
```

For a Singleton of List and Tuple...

```
>>> a_list = [3,5,8]
>>> print(a_list)
[3, 5, 8]
>>> type(a_list)
<class 'list'>
```

- a list with only one element

```
>>> b_list = [3]
>>> print(b_list)
[3]
>>> type(b_list)
<class 'list'>
>>> |
```

```
>>> a_tuple=(3,5,8)
>>> print(a_tuple)
(3, 5, 8)
>>> type(a_tuple)
<class 'tuple'>
```

- a tuple with only one element

```
>>> b_tuple=(3)
>>> print(b_tuple)
3
>>> type(b_tuple)
<class 'int'>
```

!!!

A Tuple with only one element

```
>>> b_tuple=(3)
>>> print(b_tuple)
3
>>> type(b_tuple)
<class 'int'>
```

- Correct way

```
>>> c_tuple = (3,)
>>> print(c_tuple)
(3,)
>>> type(c_tuple)
<class 'tuple'>
>>> c_tuple[0]
3
```

Note the
comma
here

But then, why use Tuple? Or List?

Or when to use Tuple? When to use
List?

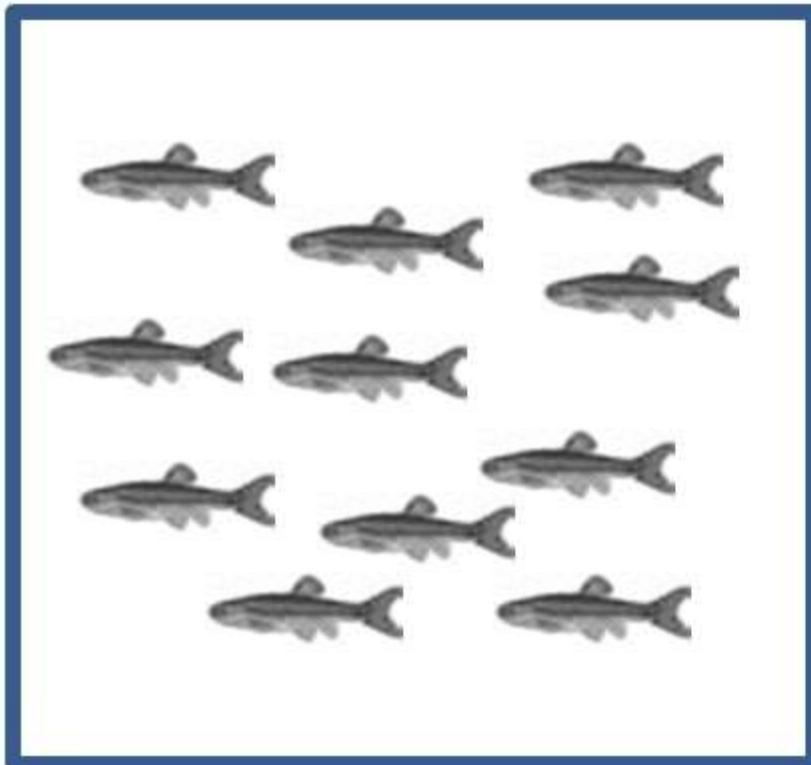
English Grammar

- Which sentence is grammatically correct?
 - “I have more than one fish. Therefore, I have many *fish*”
 - “I have more than one fish. Therefore, I have many *fishes*”
- Both of them are grammatically correct!
 - But they mean different things

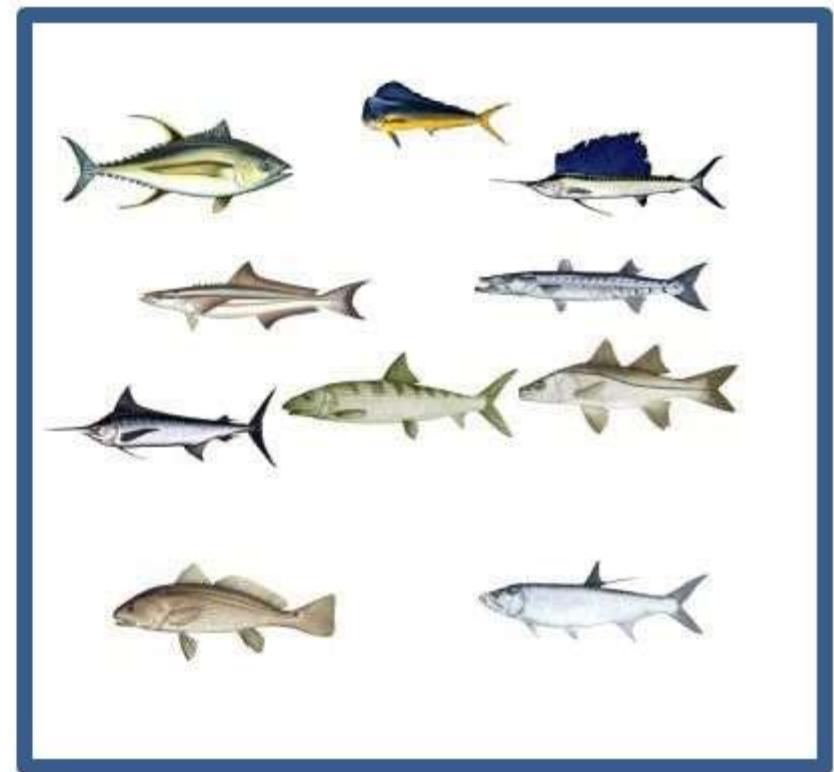
Fish vs Fishes

- The plural of fish is usually *fish*.
- When referring to more than one species of fish, especially in a scientific context, you can use *fishes* as the plural.

Fish vs. Fishes



“This tank is full of fish.”



“The ocean is full of fishes.”

List vs Tuple, Cultural Reason

- List
 - Usually stores a **large** collection of data with the **same type (homogenous)**
 - E.g. List of 200 student names in a class
- Tuple
 - Usually stores a **small** collections of items with **various data types/concepts (heterogeneous)**
 - E.g. A single student record with name (string), student number(string) and mark(integer)

But, violating this “culture” will NOT cause any syntax error

An Example

- To store the data on a map
 - These are the locations of **100** nice restaurants in Singapore
 - The location of each restaurant is recorded as the coordinates value of x and y
 - (100,50)
 - (30, 90)
 - (50, 99)
 - etc...

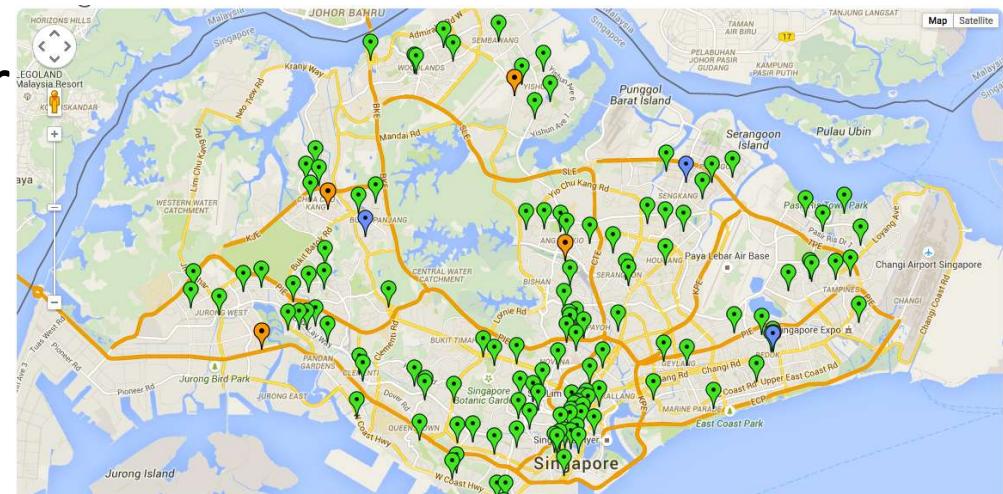


An Example

- I will code like this

```
locations_of_nice_restaurants = [ (100, 50),  
                                  (30, 90), (50, 90) ]
```

- Is it
 - 1. a tuple of tuples,
 - 2. a tuple of lists,
 - 3. a list of tuples, or
 - 4. a list of lists?



Find all the restaurants near me

- I will code like this

```
locations_of_nice_restaurants = [(100, 50),  
                                  (30, 90), (50, 90)]
```

shortened the name

```
def find_restaurants(my_current_pos):  
    locations = generate_list()  
    output_list = []  
  
    for loc in locations:  
        if distance(my_current_pos, loc) < DISTANCE_RANGE:  
            output_list.append(loc)  
  
    return output_list
```

```

def find_restaurants(my_current_pos):
    locations = generate_list()
    output_list = []

    for loc in locations:
        if distance(my_current_pos, loc) < DISTANCE_RANGE:
            output_list.append(loc)

    return output_list

def generate_list():
    output_list = []
    for i in range(NO_RESTAURANTS):
        output_list.append( (random.randint(1,SIZE_OF_SG),
                            random.randint(1,SIZE_OF_SG)) )
    return output_list

def distance(p1,p2):
    return sqrt( square(p1[0]-p2[0]) + square(p1[1]-p2[1]) )

def square(x):
    return x * x

```

A list

Just a fake function
to generate the list
for this demo

A tuple

```
def find_restaurants(my_current_pos):  
    locations = generate_list()  
    output_list = []  
  
    for loc in locations:  
        if distance(my_current_pos, loc) < DISTANCE_RANGE:  
            output_list.append(loc)  
  
    return output_list
```

```
>>> find_restaurants((50,50))  
[(45, 52), (59, 47), (51, 41)]  
>>> find_restaurants((50,50))  
[(55, 48), (54, 55)]  
>>> find_restaurants((50,50))  
[(51, 58), (45, 47)]  
>>> find_restaurants((50,50))  
[(43, 55), (48, 43), (43, 48), (54, 43)]
```

Challenge:
Find the nearest THREE restaurants

Instead of ALL

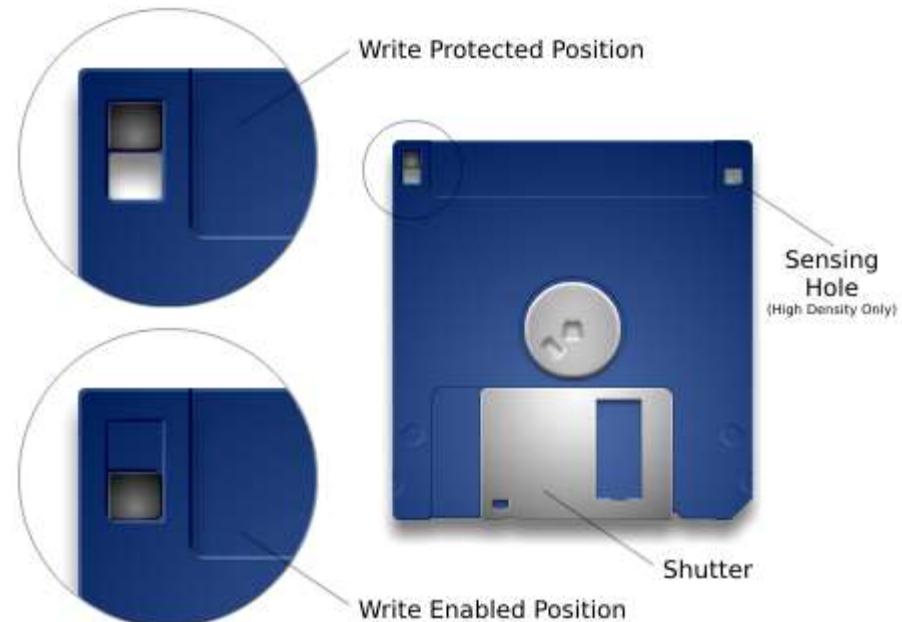
List vs Tuple, Cultural Reason

- List
 - Usually stores a **large** collection of data with the **same type (homogenous)**
 - E.g. List of 200 student names in a class
- Tuple
 - Usually stores a **small** collections of items with **various data types/concepts (heterogeneous)**
 - E.g. A single student record with name (string), student number(string) and mark(integer)

But, violating this “culture” will NOT cause any syntax error

List vs Tuple, Technical Reasons

- Immutable vs mutable
 - Tuple is **Write protected** (Immutable)
- List can be changed within a function
 - NOT passed by value
 - Mutable



Recap: Primitive Data Types

```
x = 0
```

```
def changeValue(n):  
    n = 999  
    print(n)
```

```
changeValue(x)  
print(x)
```

- The `print()` in “changeValue” will print 999
- But how about the last `print(x)?`
 - Will x becomes 999?
- (So actually this function will NOT change the value of x)

Recap: Primitive Data Types

```
x = 0
```

```
def changeValue(n):  
    n = 999  
    print(n)
```

```
changeValue(x)  
print(x)
```

- n is another copy of x
- You can deem it as

```
def changeValue(x):  
    n = x  
    n = 999  
    print(n)
```

Pass By Values

But for List

- Mutable!

```
>>> l = [1, 2, 3]
>>> changeSec(l)
```

Inside function

```
[1, 'changed!', 3]
>>> print(l)
[1, 'changed!', 3]
```

```
def changeSec(a):
    a[1] = 'changed!'
    print('Inside function')
    print(a)
```

!!!

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Sets

- A set is an **unordered** collection of **immutable** elements with **no duplicate** elements
 - Unordered: You cannot get a single element by its index like `s[2]`
 - No duplicate: every element exists only once in a set

```
>>> set1 = {1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3}  
>>> set1  
{1, 2, 3, 4, 5, 6, 7, 8}
```

Tuples use '()' and ''
Lists use '[]' and ''
Sets use '{}' and ''

Python
Removes
duplicates
for you

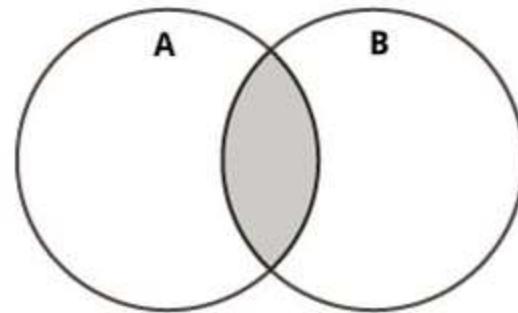
Sets

- Some operations are not available because sets are NOT indexed

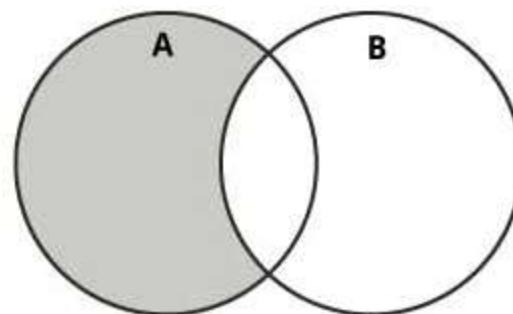
a[i]	return i-th element of a
a[i:j]	returns elements : up to j - 1
len(a)	returns numbers of elements in sequence
min(a)	returns smallest value in sequence
max(a)	returns largest value in sequence
x in a	returns True if x is a part of a
a + b	concatenates a and b
n * a	creates n copies of sequence a

Set Operations

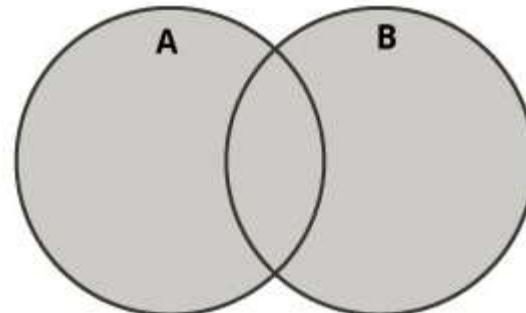
- Intersection



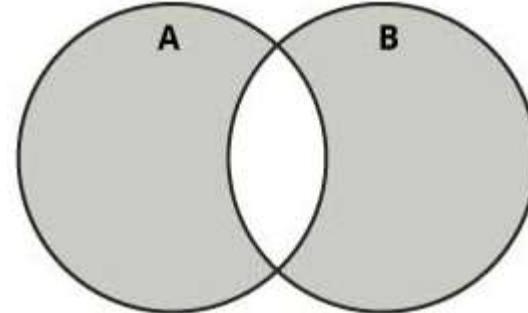
- $A - B$



- Union



- Symmetric Difference



Sets

- Usual set operations

```
>>> setA = {1,2,3,4}  
>>> setB = {3,4,5,6}  
>>> setA | setB ← Union  
{1, 2, 3, 4, 5, 6}  
>>> setA & setB ← Intersection  
{3, 4}  
>>> setA - setB ← A - B  
{1, 2}  
>>> setA ^ setB ← (A | B) - A & B  
{1, 2, 5, 6}
```

Sets

```
>>> setA.remove(1) ← Remove like a list
>>> setA
{2, 3, 4}
>>> setA.remove(1) ← But error if element missing
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    setA.remove(1)
KeyError: 1
>>> setA.discard(1) ← But we can use
>>> | discard instead
```

Sets are Iterable

```
my_set = {1,2,3}          1  
for i in my_set:         2  
    print(i)              3
```

Set from List and Vice-Versa

```
>>> my_list = [1,2,3]
>>> my_set = set(my_list)
>>> my_set
{1, 2, 3}
```

```
>>> my_set = {4,5,6}
>>> my_list = list(my_set)
>>> my_list
[4, 5, 6]
```

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Dictionary

e•merge (i-mûrj') *v.* **e•merged**, **e•merg•ing**.

1. To rise up or come forth into view; appear. 2. To come into existence. 3. To become known or evident. [Lat. *emergere*.] —**e•mer'gence** *n.* —**e•mer'gent** *adj.*

e•merg•en•cy (i-mûr'jân-sé) *n., pl. -ies.* An unexpected situation or occurrence that demands immediate attention.

e•mer•itus (i-mér'i-tës) *adj.* Retired but retaining an honorary title: *a professor emeritus.* [Lat., p.p. of *emereri*, to earn by service.]

em•er•ry (ém'ə-rë, əm'rë) *n.* A fine-grained impure corundum used for grinding and polishing. [< Gk *smuris*.]

emet•ic (i-mét'ik) *adj.* Causing vomiting. [< Gk. *emein*, to vomit.] —**emet'ic**, *n.*

-emia suff. Blood: *leukemia.* [< Gk. *haima*, blood.]

em•i•grate (ém'i-grät') *v.* **-grat•ed**, **-grat•ing**. To leave one country or region to settle in another. [Lat. *emigrare*.] —**em'i-grant** *n.* —**em'i-gra'tion** *n.*

é•mi•gré (ém'i-grá') *n.* An emigrant, esp. a refugee from a revolution. [Fr.]

em•i•nence (ém'ə-nëns) *n.* 1. a position of great distinction or superiority. 2. A rise or elevation of ground; hill.

em•i•nent (ém'ə-nënt) *adj.* 1. Outstanding, as in reputation; distinguished. 2. Towering above others; projecting. [< Lat. *eminere*, to stand out.] —**em'i•nent•ly** *adv.*

Word

em•phatic (ém-fät'ik) *adj.* Expressed or performed with emphasis. [< Gk. *emphatikos*.] —**em•phat'ic•al•ly** *adv.*

em•phy•se•ma (ém'fi-sé'mä) *n.* A disease in which the air sacs of the lungs lose their elasticity, resulting in an often severe loss of breathing ability. [< Gk. *emphusēma*.]

em•pire (ém'pîr') *n.* 1. A political unit, usu. larger than a kingdom and often comprising a number of territories or nations, ruled by a single central authority. 2. Imperial dominion, power, or authority. [< Lat. *imperium*.]

em•pir•i•cal (ém-pir'i-käl) *adj.* Also **em•pir•ic** (-pir'ik). 1. Based on observation or experiment. 2. Relying on practical experience rather than theory. [< Gk. *empeirikos*, experienced.] —**em•pir'ic•al•ly** *adv.*

em•pir•i•cism (ém-pir'i-siz'më) *n.* 1. The view that experience, esp. of the senses, is the only source of knowledge. 2. The employment of empirical methods, as in science.—**em•pir'ic•ist** *n.*

em•place•ment (ém-pläs'mënt) *n.* 1. A prepared position for guns within a fortification. 2. Placement. [Fr.]

em•ploy (ém-ploï') *v.* 1. To engage or use the services of. 2. To put to service; use. 3. To devote or apply (one's time or energies) to an activity. —*n.* Employment. [< Lat. *implicare*, to involve.] —**em•ploy'a•ble** *adj.*

em•ploy•ee (ém-ploï'ë, əm'ploi-ë') *n.* Also **em•ploy•e**. One who works for another.

Its meaning

Word

Its meaning

â pat â pay â care â father ē pet ē be ī pit ī tie ī pier ò pot ò toe ò paw, for òi noise
oo took oo boot ou out th thin th this ù cut ú urge yoo abuse zh vision ɔ about, item,
edible, gallop, circus

Dictionary

- You search for the word in the dictionary
- Then look for its meaning



- Each word has a **correspondent** meaning

Python Dictionary

- You search for the **key** in the dictionary
- Then look for its **value**



- Each key has a **correspondent** value

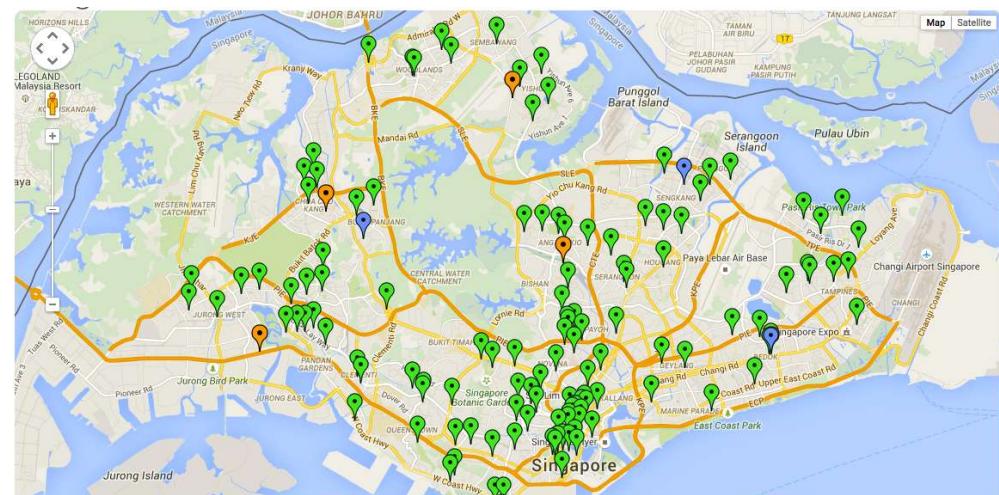
```
>>> students = {'A100000X': 'John', 'A123456X': 'Peter',  
'A999999X': 'Paul'}  
>>> students['A123456X']  
'Peter'
```

key : value
pair

Tuples use '(' and ')'
Lists use '[' and ']'
Sets and Dict use '{' and '}'

An Example

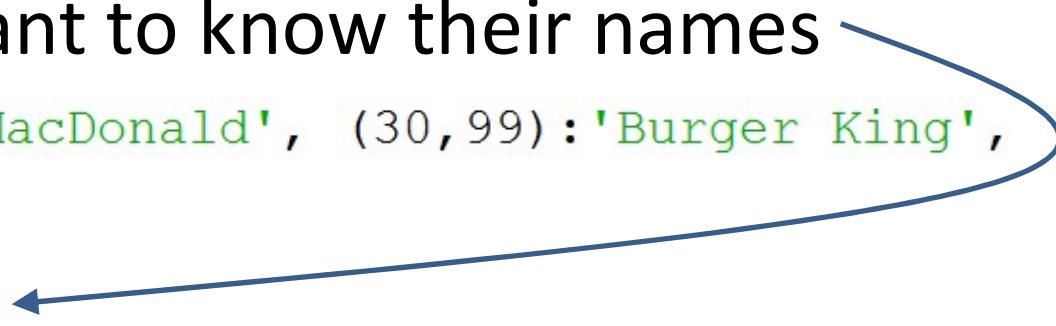
- To store the data on a map
 - These are the locations of **100** nice restaurants in Singapore
 - The location of each restaurant is recorded as the coordinates value of x and y **and name**
 - (10,20):Pizza Hut



Python Dictionary

- Key: location
- Value: restaurant name
- After you searched for the nearest restaurants, you want to know their names

```
>>> locations = { (10, 30) : 'MacDonald', (30, 99) : 'Burger King',  
(22, 33) : 'Pizza Hut'}  
>>>  
>>> locations[(22, 33)]  
'Pizza Hut'
```



Recap: List

- Or tuples

```
>>> vm = ['M&M', 'Twix', 'Milky Way', 'Oreo']  
>>> vm[1]  
'Twix'
```



Index:
From 0 to $\text{len}(a)-1$

Input a number



Output an item



But when you go to Japan

- You are not inputting a number (index)!

```
>>> vmj = {'Beef noodle small':290, 'Beef noodle big':390}  
>>> vmj['Beef noodle small']  
290
```

~~Input a number~~ a name



Output an item



To set up a dictionary

- Each pair has a key and a value

```
>>> vmj = {'Beef noodle small':290, 'Beef noodle big':390}
```

The code defines a dictionary named `vmj` with two entries. Each entry consists of a key (a string) and a value (an integer). The keys are 'Beef noodle small' and 'Beef noodle big'. The values are 290 and 390 respectively. The diagram uses blue curly braces to group the key and value of each entry together, with the word 'key' below the first brace and 'value' below the second.

What is Dictionary?

- Key is on the left, Value on the right

```
>>> my_dictionary = {'a':1, 'b':2}  
>>> my_dictionary['b']  
2
```

- Summary: A data structure used for
“When I give you X, give me Y”
- Can store any type
- Called HashTable in some other languages

How is a Dictionary Useful?

- Keep Track of Things by Key!
 - Eg, keeping track of stocks of fruits

```
my_stock = {"apples":450,"oranges":412}  
my_stock["apples"]  
>>> 450
```

```
my_stock["apples"] + my_stock["oranges"]  
>>> 862
```

How is a Dictionary Useful?

- Keep Track of Things by Key!
 - When you want to get an associated operation
(eg, alphabets to numeric integers)

```
my_alphabet_index = {'a':1,'b':2... 'z':26}
my_alphabet_index['z']
>>> 26
```

Dictionary Methods

- Access (VERY FAST! - Almost instant!)
- Assignment
- Removal
- Other Dictionary Methods

Dictionary Access

```
>>> my_fruit_inventory = {"apples":450,"oranges":200}  
>>> my_fruit_inventory["apples"]  
450  
>>> my_fruit_inventory.get("apples")  
450  
>>> my_fruit_inventory["pears"]  
KeyError!  
>>> my_fruit_inventory.get("pears")  
None
```

****Cannot access keys which don't exist!****

- Accessing with [] will crash if does not exist
- Accessing with .get() will NOT crash if key does not exist

Dictionary Assignment

```
>>> my_fruit_inventory[“pears”] = 100  
>>> print(my_fruit_inventory)  
{“apples”:450, “oranges”:200, “pears”:100}
```

- Caution: This OVERWRITES existing values!

```
>>> my_fruit_inventory[“oranges”] = 100  
>>> print(my_fruit_inventory)  
{“apples”:450, “oranges”:100, “pears”:100}
```

Dictionary Removal

```
>>> my_fruit_inventory =  
{"apples":450,"oranges":200}
```

```
>>> my_fruit_inventory.pop("apples")  
>>> print(my_fruit_inventory)  
{'oranges':200}
```

- OR

```
>>> del my_fruit_inventory["apples"]
```

Other Dictionary Methods

- | | |
|-----------|----------------------------|
| .clear() | • clear all |
| .copy() | • make a copy |
| .keys() | • return all keys |
| .values() | • return all values |
| .items() | • return all keys + values |

Dictionary is Iterable

```
my_dict = {'a':1, 'b':2}
```

```
for key in my_dict:  
    print(key)
```

```
for key in my_dict:  
    print(key, my_dict[key])
```

```
a  
b  
a 1  
b 2  
a 1  
b 2
```

```
for key, value in my_dict.items():  
    print(key, value)
```

Sequence in Python

- Indexed
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary