

# Data Collections (Sequences)

It's complicated

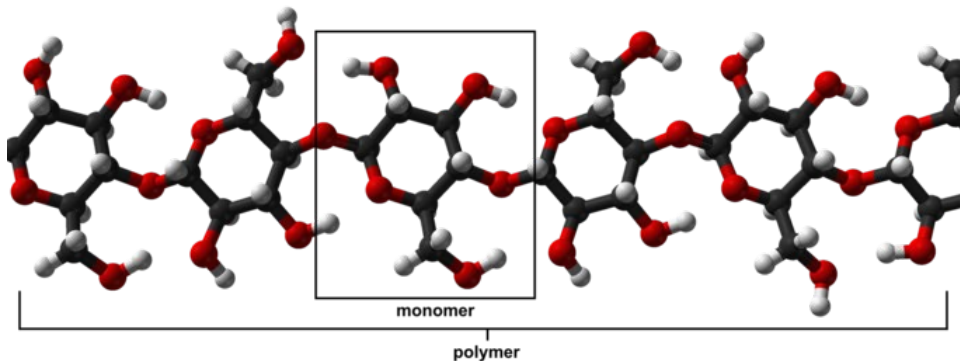
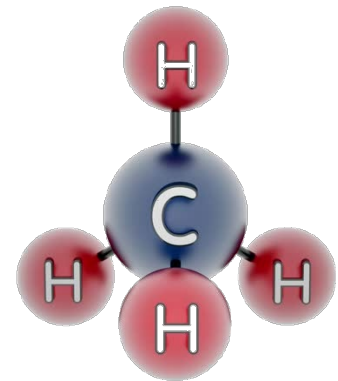
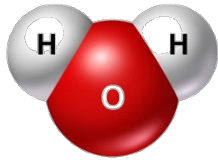
# Primitives vs Structures (Chemistry)

## Primitives

- Hydrogen atom
- Oxygen atom
- etc

## Structure

- Water molecules
- Methane
- Polymer



# Primitives vs Structure (Python)

## Primitive

- Integers
- Boolean
- Float

## Structure

- A rational number  $\frac{a}{b}$ 
  - Two integers
- Student record in a course
  - Student name
  - Student number
  - Grades
- Sequence
  - e.g. all the marks in a class
- Strings
- Sets

# Compound Data

- You can store the mark of a single student
  - peter\_score = 100
- But, how do you store the marks of a class with 50 students?
  - student1\_score = 100
  - student2\_score = 89
  - student3\_score = 70
  - student4\_score = 79
  - ...

# An Example



- To store the data on a map
  - We have the locations of **100** nice restaurants in Singapore
  - Then, you want to list out the 10 most nearest restaurants that are nearest to you



# 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...

How to store all these locations?



# Sequence

- A collection of “something”
  - E.g. A collection of motions



# Sequences in Python

- Strings
- Lists
- Tuples
- Others:
  - Sets
  - Dictionary



See if we  
have time  
today




# Recap: Strings

- Strings are **sequences** of characters


```
>>> name = 'Alan'
>>> course_code = 'IT1007'
>>> print(course_code)
IT1007
>>> course_code[2]
```

Index



```
>>> s = 'abcdef'
>>> print('c' in s)
True
>>> print('z' in s)
False
```

Is the character 'c' in the string s?



	a	b	c	d	e	f
Index	0	1	2	3	4	5

# String Slicing

Non-inclusive



Default  
start = 0  
stop = #letters  
step = 1

**s[start:stop:step]**

```
>>> s = 'abcdef'
```

```
>>> s[0:2]
```

```
'ab'
```

```
>>> s[1:2]
```

```
'b'
```

```
>>> s[:2]
```

```
'ab'
```

```
>>> s[1:5:3]
```

```
'be'
```

```
>>> s[::2]
```

```
'ace'
```

```
>>> s[::-1]
```

```
???
```

# All Indexed Sequences can...

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

# String Example

```
>>> s1 = 'Minions like bananas '
```

```
>>> s1[5]  
'n'
```

```
>>> s1[0:6]  
'Minion'
```

```
>>> len(s1)  
21
```

```
>>> max(s1)  
's'
```

```
>>> min(s1)  
' '
```

```
>>> 'o' in s1  
True
```

```
>>> 'z' in s1  
False
```

```
>>> s1 + 'and Gru'  
'Minions like bananas and Gru'
```

```
>>> s1 * 3  
'Minions like bananas Minions like bananas Minions like bananas '
```

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

# Sequence in Python

- Strings
- Lists
- Tuples

# List

- Strings are **sequences** of characters
- Lists are **sequences** of **anything**

```
>>>
>>> even_numbers_10 = [0, 2, 4, 6, 8, 10]
>>> my_good_friends = ['Peter', 'Paul', 'Mary']
>>> ans_to_universe = ['Nothing', 'Deity', 42, True, None]
>>> ans_to_universe[3:5]
[True, None]
>>> len(ans_to_universe)
5
```

slicing

Can be more  
than one type



answer to life the universe and everything



All

Images

Maps

Videos

News

More

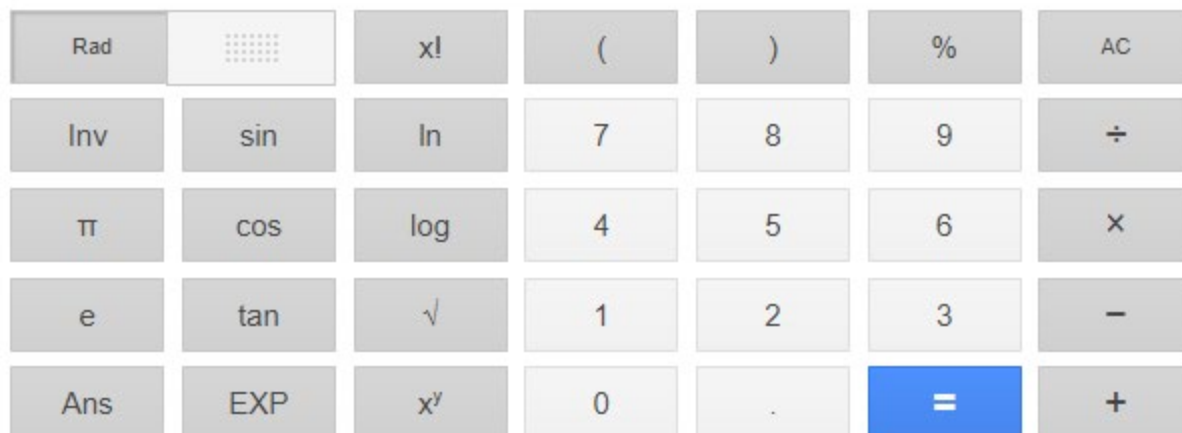
Settings

Tools

About 32,300,000 results (0.58 seconds)

answer to life the universe and everything =

42



```
>>>
>>> even_numbers_10 = [0, 2, 4, 6, 8, 10]
>>> my_good_friends = ['Peter', 'Paul', 'Mary']
>>> ans_to_universe = ['Nothing', 'Deity', 42, True, None]
>>> ans_to_universe[3:5]
[True, None]
>>> len(ans_to_universe)
5
```

```
>>> type(ans_to_universe)
<class 'list'>
>>> type(ans_to_universe[0])
<class 'str'>
>>> type(ans_to_universe[2])
<class 'int'>
>>> type(ans_to_universe[4])
<class 'NoneType'>
```



# All Indexed Sequences can...

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

```
>>> even_numbers_10 + my_good_friends + ans_to_universe  
[0, 2, 4, 6, 8, 10, 'Peter', 'Paul', 'Mary', 'Nothing',  
'Deity', 42, True, None]
```

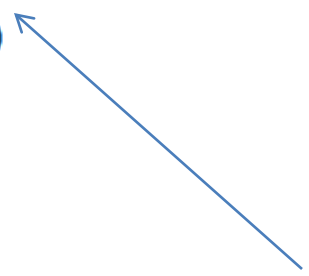
# On Top of the Common Features

- Can **Append** and **Remove**

- Add/delete an element

```
>>> my_good_friends.append('John')
>>> print(my_good_friends)
['Peter', 'Paul', 'Mary', 'John']

>>> my_good_friends.remove('Paul')
>>> print(my_good_friends)
['Peter', 'Mary', 'John']
```



Error if the element  
does not exist in the  
list

# On Top of the Common Features

- Can **Append** and **Remove**
  - Add/delete an element
  - But how about this? How many '2' will be removed?

```
>>> a_list = [1,2,3,4,1,2,3,4]
>>> a_list.remove(2)
>>> a_list
[1, 3, 4, 1, 2, 3, 4]
```

- Only the first appearance of '2' will be removed
- How about removing an item NOT in the list?
  - Error!

# What if...

```
>>> my_good_friends.append(even_numbers_10)
>>> print(my good friends)
```

- Which one is the correct output?

```
['Peter', 'Mary', 'John', [0, 2, 4, 6, 8, 10]]
```



- or

```
['Peter', 'Mary', 'John', 0, 2, 4, 6, 8, 10]
```

– This is the result of

```
>>> my_good_friends + even_numbers_10
```

# Difference between

- Append a list to a list



- Concatenate a list to a list



+



=



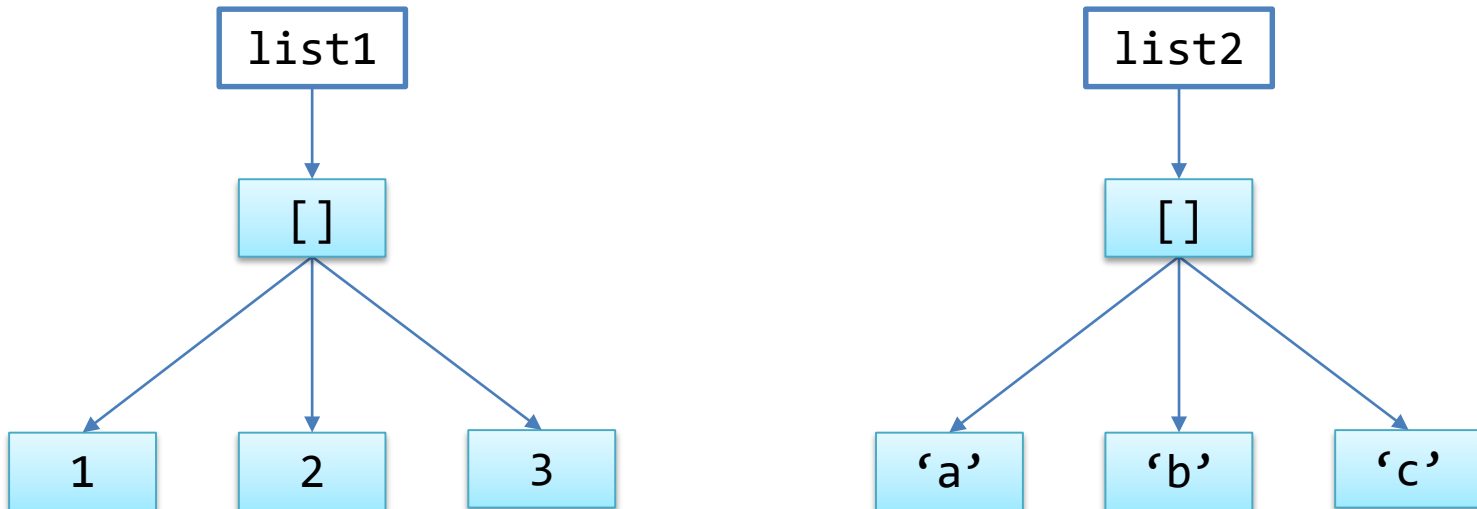
# Lists of Anything

- 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]]
```

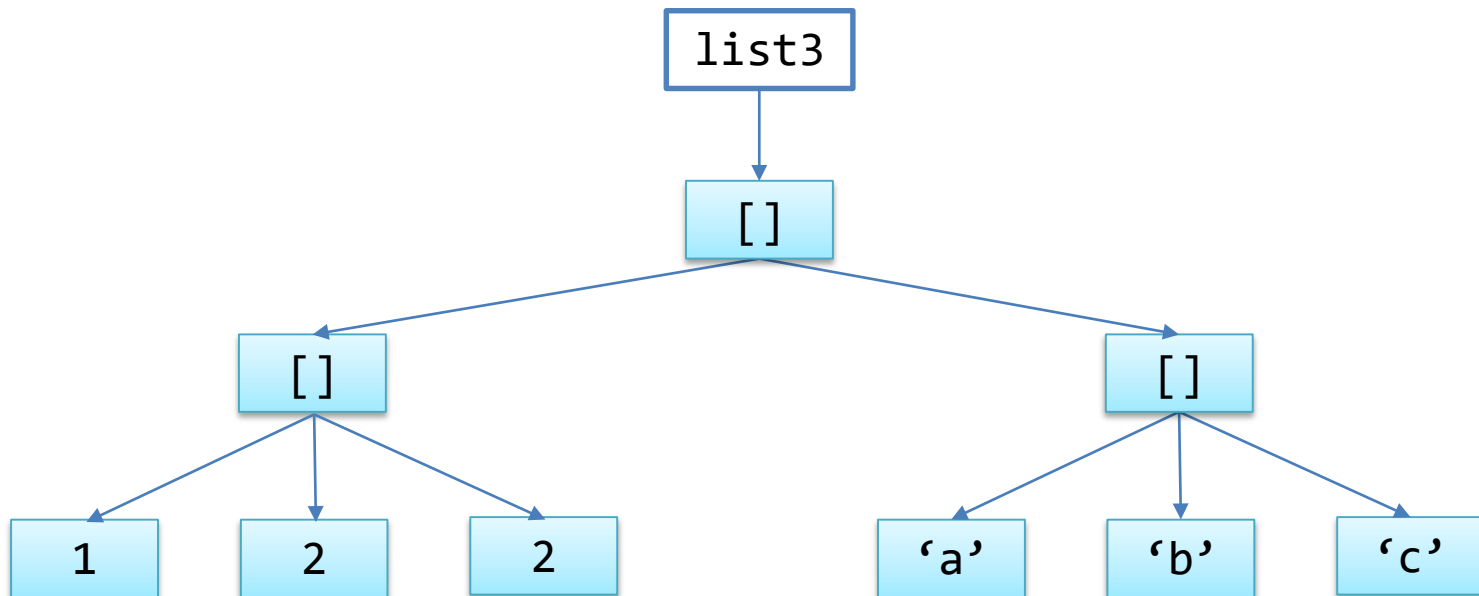
# Block Diagram

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



# Block Diagram

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





# Iterables

```
>>> s = 'abcde'  
>>> for i in s:  
    print(i)
```

```
a  
b  
c  
d  
e
```

# 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

```
>>> for i in ans_to_universe:  
    print (i)
```

Nothing

Deity

42

True

None

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

0

1

2

3

4

# Iterables

- anything that can be looped over
  - E.g. you can loop over a string
- anything that can appear on the right-side of a for-loop

```
>>> for i in ans_to_universe:  
    print (i)
```

```
Nothing  
Deity  
42  
True  
None
```

```
for x in iterables:  
    do something about x
```

```
>>> ans = 0  
>>> for i in even_numbers_10:  
    ans += i
```

```
>>> print(ans)  
30
```

# 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]
```

# Conversion between Strings and Lists

- Remember we can convert an integer to string, or vice versa

```
>>> str(123)
'123'
>>> int('123')
123
```


- What happen when we convert a string to a list?

```
>>> list('123')
['1', '2', '3']
```

- And reverse?

```
>>> str([1, 2, 3])
'[1, 2, 3]'
```

Note that it  
won't become  
'123'



# 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  $< 50$

*i is from 2 to 7  
(7 = sqrt(50))*

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

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

[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  $< 50$

*i* is from 2 to 7

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

```
>>> 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, 10, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 21, 28, 35, 42, 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]
```

# Sequence in Python

- Strings
- Lists
- Tuples

All iterables

```
>>> s = 'abcde'  
>>> for i in s:  
    print(i)
```

```
a  
b  
c  
d  
e
```



# Tuple

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

```
>>> a_tuple = (12, 13, 'dog')
```

```
>>> a_tuple[1]
```

```
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):
```

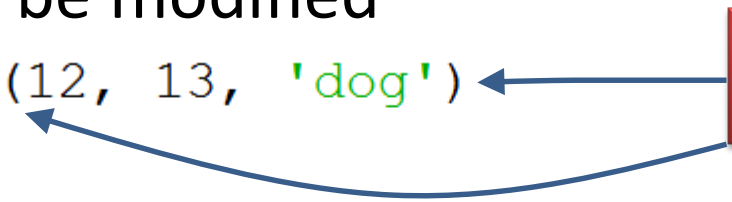
```
File "<pyshell#131>", line 1, in <module>
```

```
    a_tuple.append(1)
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

```
>>>
```

Tuples use '(' and ')'  
Lists use '[' and ']'



# 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?

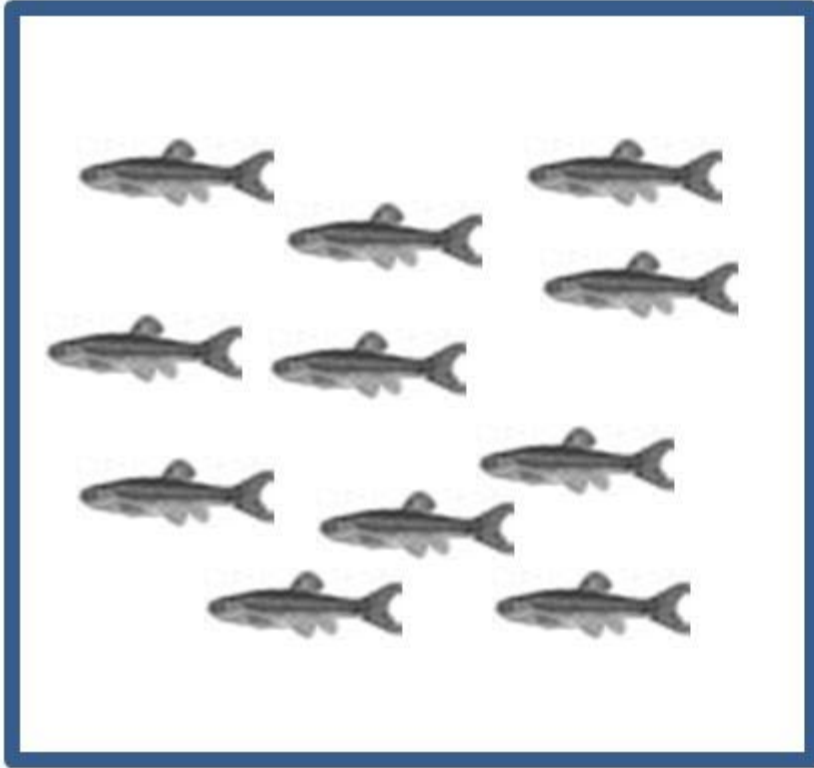
# English Grammar

- 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

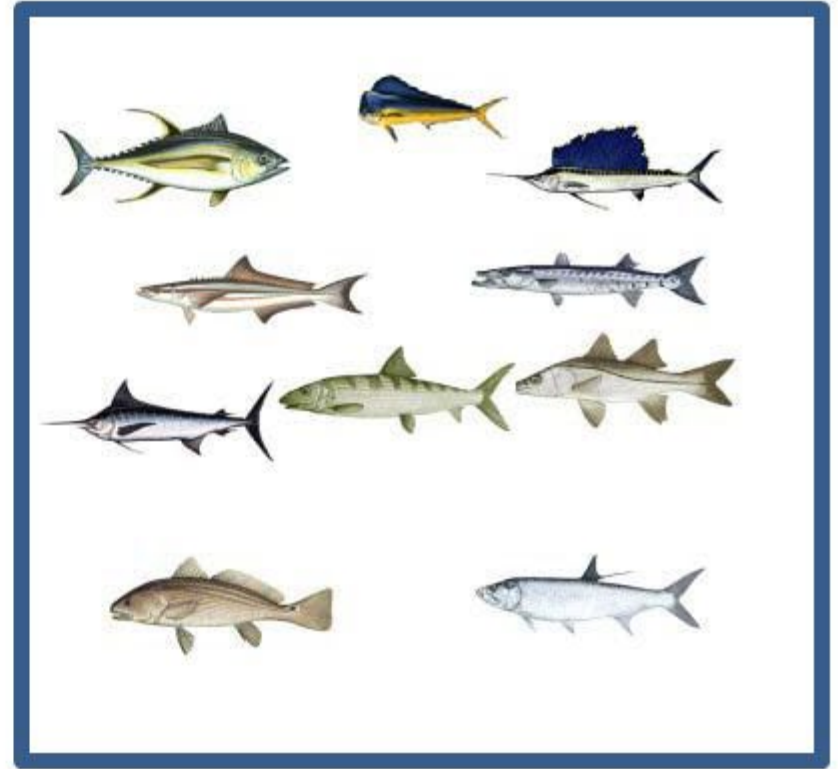
# Fish vs Fishes

- 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)]
```

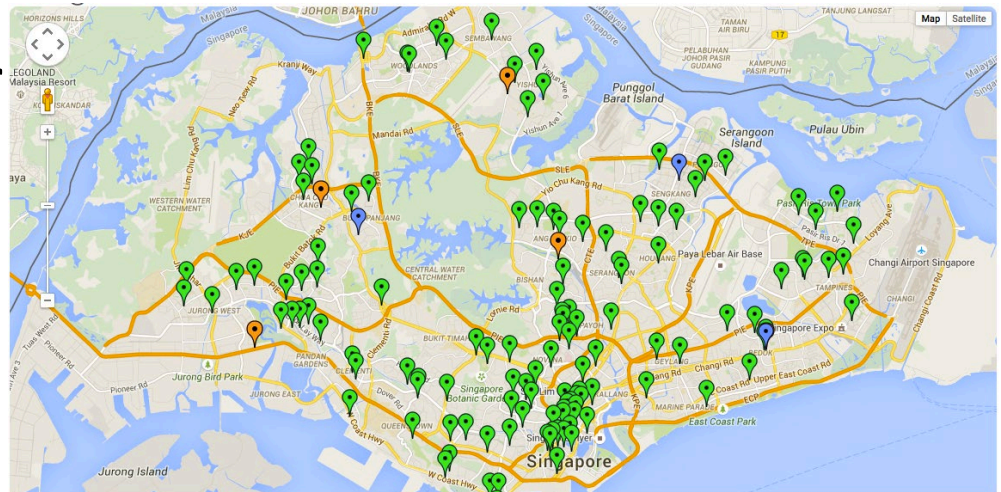
- Is it

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

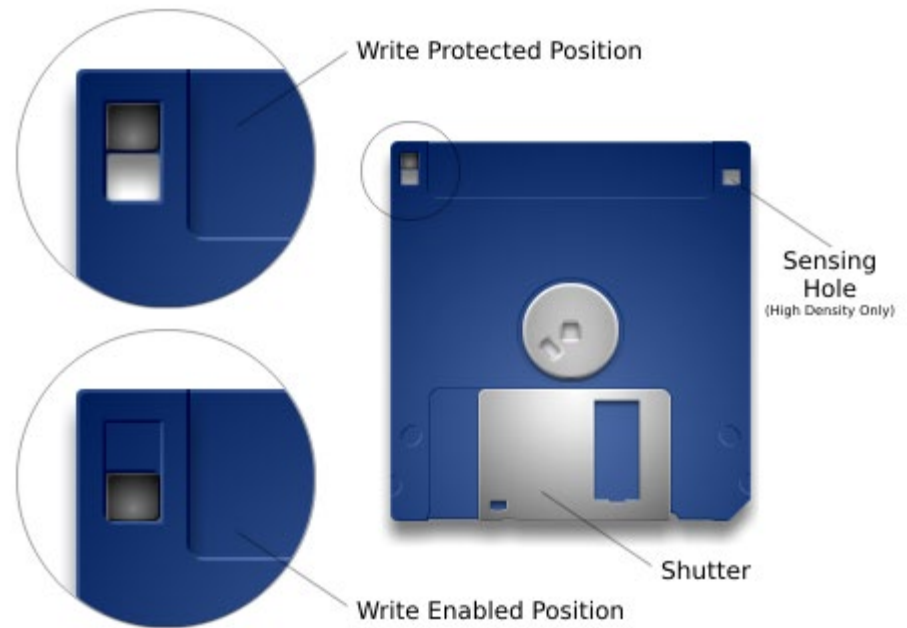
- List
  - Usually stores a **large** collection of data with the **same type (homogenous)**
  - E.g. List of 200 student names in a class
- Tuple
  - Usually stores a **small** collections of items with **various data types/concepts (heterogeneous )**
  - E.g. A single student record with name (string),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

```
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: Pass by Values

```
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)
```

# 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

- Strings
- Lists
- Tuples

All iterables  
Because the  
elements are  
indexed

```
>>> s = 'abcde'  
>>> for i in s:  
    print(i)
```

a  
b  
c  
d  
e

- Non-indexed collection:
  - Sets
  - Dictionary

# Sets

- A set is an **unordered** collection 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

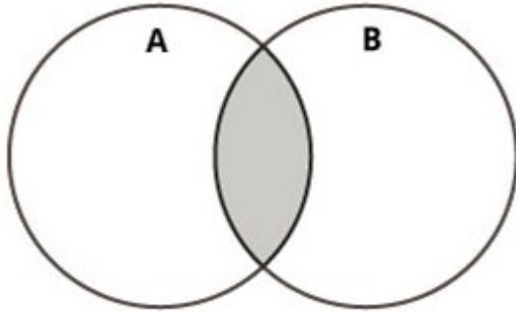
# Sets

- Some operations are not available because sets are NOT indexed

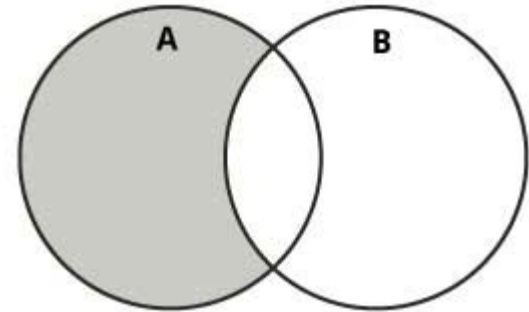
<del>a[i]</del>	<del>return i-th element of a</del>
<del>a[i:j]</del>	<del>returns elements i up to j - 1</del>
len(a)	returns numbers of elements in sequence
min(a)	returns smallest value in sequence
max(a)	returns largest value in sequence
x in a	returns True if x is a part of a
<del>a + b</del>	<del>concatenates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

# Set Operations

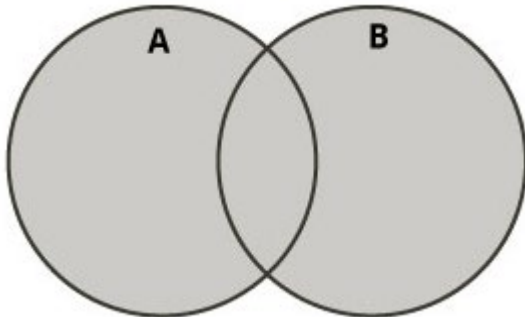
- Intersection



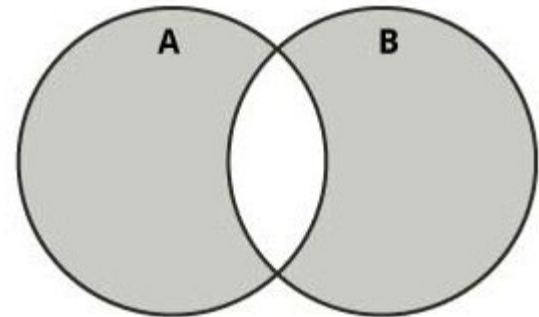
- $A - B$



- 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
```

# Sequence in Python

- Strings
  - Lists
  - Tuples
- 
- All iterables  
Because the  
elements are  
indexed

- Non-indexed collection:
  - Sets
  - Dictionary

# Dictionary

Word

**e•merge** (ī-mûrj') *v.* **e•merged**, **e•merging**.

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•mer•gen•cy** (ī-mûr'jən-sē) *n., pl. -ies.* An unexpected situation or occurrence that demands immediate attention.

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

**e•mer•y** (ēm'ə-rē, ēm'rē) *n.* A fine-grained impure corundum used for grinding and polishing. [< Gk *smuris*.]

**e•met•ic** (ī-mēt'ik) *adj.* Causing vomiting. [< Gk. *emein*, to vomit.] —**e•met'ic**, *n.*

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

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

**e•m•i•gré** (ēm'i-grā') *n.* An emigrant, esp. a refugee from a revolution. [Fr.]

**e•m•i•nence** (ēm'ə-nəns) *n.* 1. a position of great distinction or superiority. 2. A rise or elevation of ground; hill.

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

**e•m•phat•ic** (ēm-fāt'ik) *adj.* Expressed or performed with emphasis. [< Gk. *emphatikos*.] —**e•m•phat'ic•al•ly** *adv.*

**e•m•phy•se•ma** (ēm'fi-sē'mə) *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*.]

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

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

**e•m•pir•i•cism** (ēm-pīr'i-sīz'əm) *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. —**e•m•pir'icist** *n.*

**e•m•place•ment** (ēm-plās'mənt) *n.* 1. A prepared position for guns within a fortification. 2. Placement. [Fr.]

**e•m•ploy** (ēm-ploi') *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.] —**e•m•ploy'a•ble** *adj.*

**e•m•ploy•ee** (ēm-ploi'ē, ēm'ploi-ē') *n.* Also **e•m•ploy•e**. One who works for another.

Its meaning

Word

Its meaning

ă pat ā pay â care ă father ě pet ě be ĭ pit ī tie î pier ò pot ô toe ô paw, for oi noise  
ōō took ōō boot ou out th thin th this ũ cut ú urge yoo abuse zh vision ă 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'
```

The code snippet shows a dictionary named 'students' with three key-value pairs. The key 'A123456X' is highlighted in yellow. An arrow points from the text 'key : value pair' to this pair. Another arrow points from a red box containing a list of symbols to the curly braces used in the dictionary definition.

key : value  
pair

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

# An Example

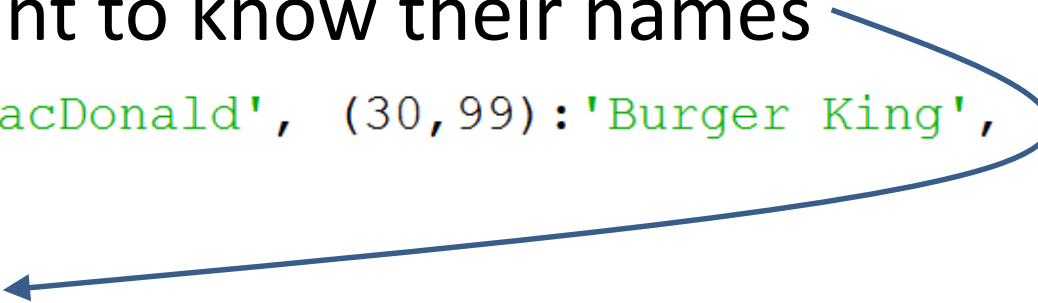
- To store the data on a map
  - These are the locations of **100** nice restaurants in Singapore
  - The location of each restaurant is recorded as the coordinates value of x and y **and name**
  - (10,20):Pizza Hut



# 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 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



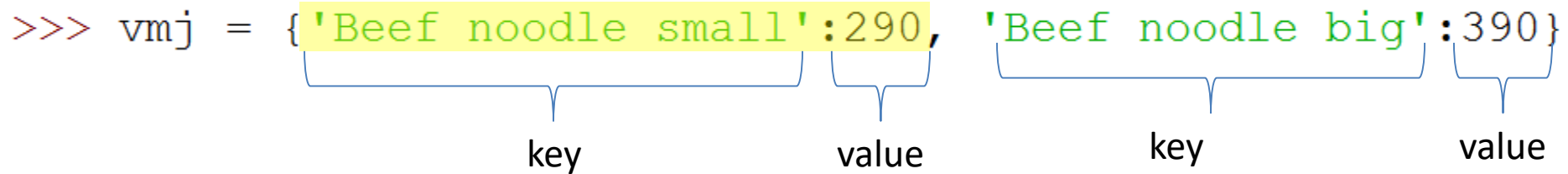
## Output an item



# To set up a dictionary

- Each pair has a key and a value

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



The diagram illustrates the structure of a Python dictionary. It shows two key-value pairs: `'Beef noodle small':290` and `'Beef noodle big':390`. Blue brackets are used to group the key and value parts of each pair. The first bracket under the first pair is labeled "key", and the second bracket is labeled "value". Similarly, the third bracket under the second pair is labeled "key", and the fourth bracket is labeled "value".

key	value
'Beef noodle small'	290
'Beef noodle big'	390

# 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

# Sequence in Python

- Indexed
  - Strings
  - Lists
  - Tuples

All iterables  
Because the  
elements are  
indexed
- Non-indexed collection:
  - Sets
  - Dictionary