Data Structure

It’s complicated

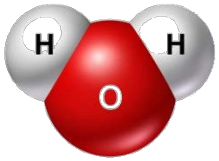
Primitives vs Structures (Chemistry)

Primitives

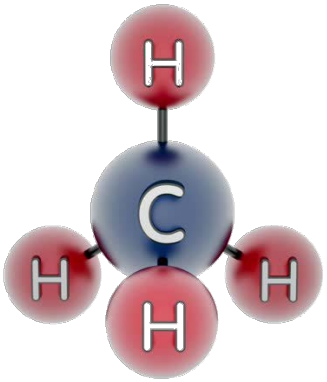
- Hydrogen atom
- Oxygen atom
- etc

Structure

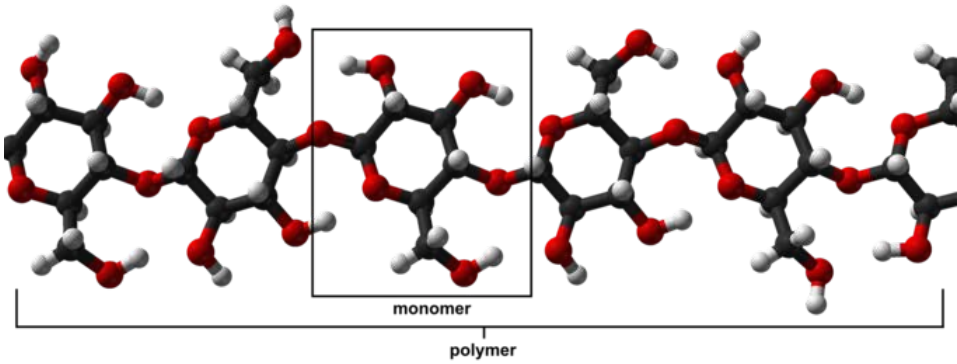
- Water molecules



- 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
  - …

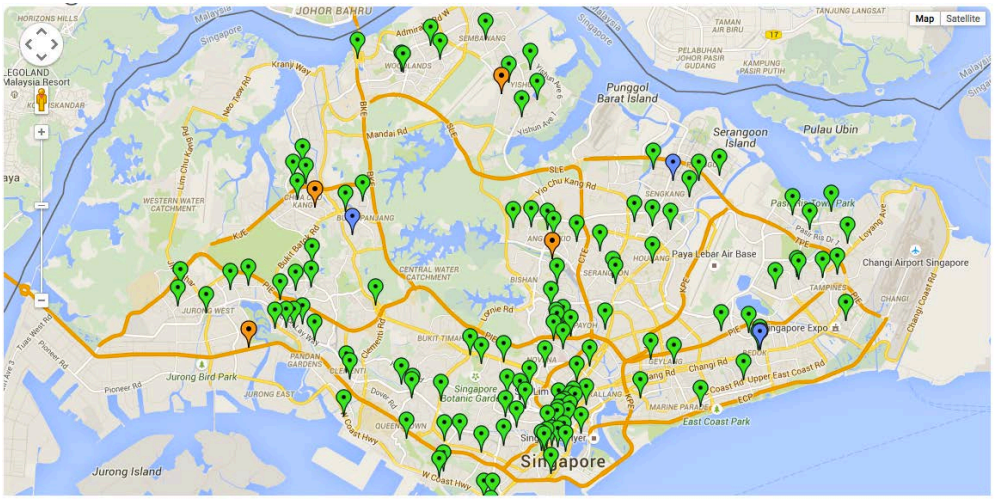
An Example

- 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

Recap: Strings

- Stings are sequences of characters

```
>>> name = 'Alan'
>>> course_code = 'IT1007'
>>> print(course_code)
IT1007
>>> course_code[2]
'1'

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

Index

Is the character ‘c’ in the string s?

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

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

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 Min
ions like bananas '
```

`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

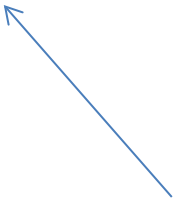
Sequence in Python

- Strings
- Lists
- Tuples

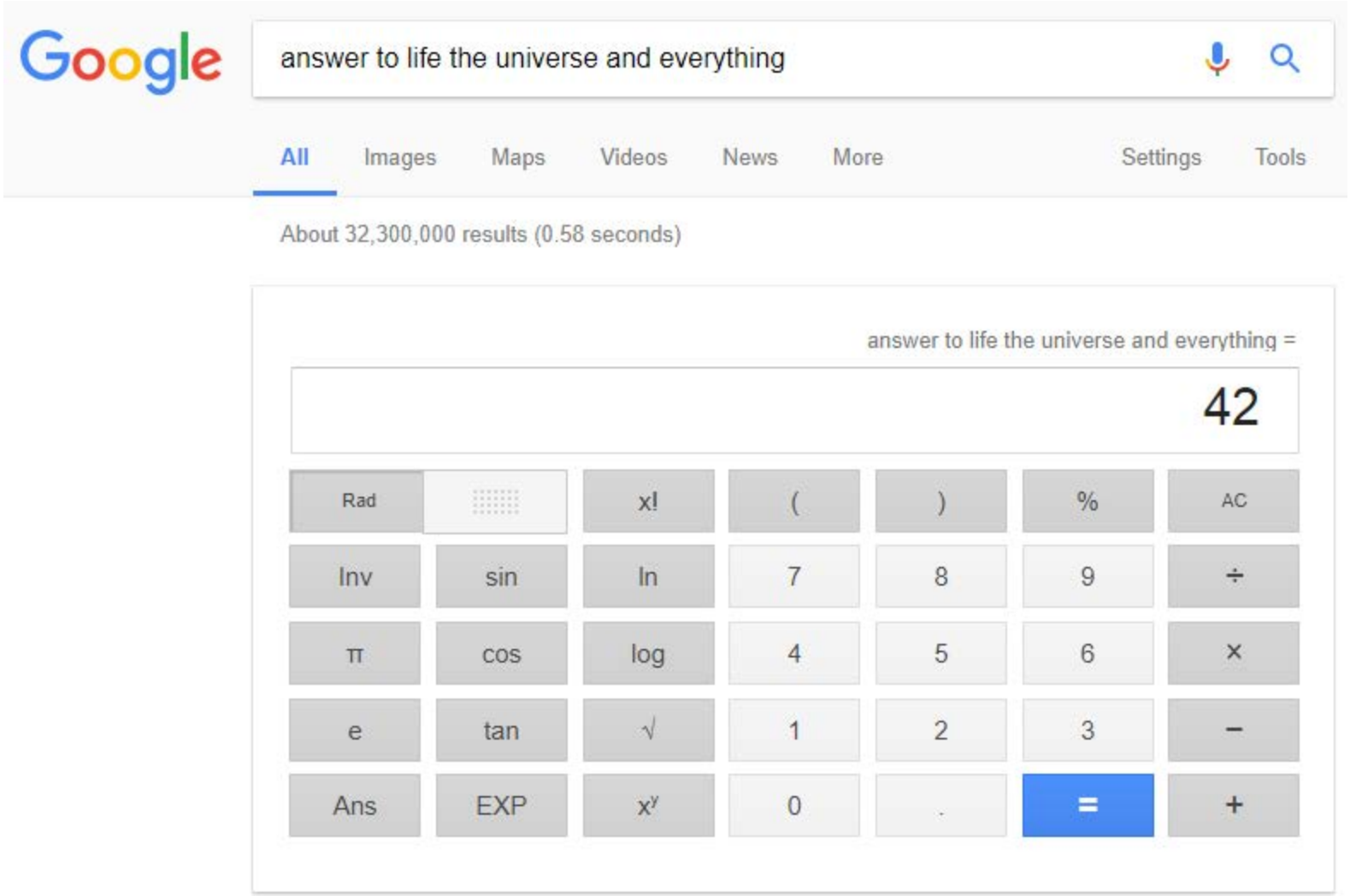
List

- 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
```



Can be more than one type


Google
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
Rad
x!
(
)
%
AC
Inv
sin
ln
7
8
9
÷
π
cos
log
4
5
6
×
e
tan
√
1
2
3
−
Ans
EXP
x^y
0
.
=
+

```
>>>
>>> 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…

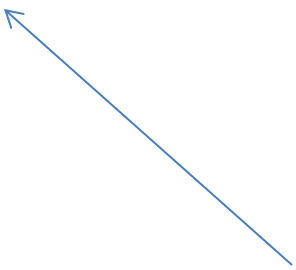
`a[i]`	return i-th element of a
`a[i:j]`	returns elements i up to j-1
`len(a)`	returns numbers of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns True if x is a part of a
`a + b`	concatenates a and b
`n * a`	creates n copies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

```
>>> 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?
    - You try

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
```

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 x in iterables:
    do something about x
```

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


Nothing
Deity
42
True
None


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


>>> print(ans)
30
```

List Comprehension

- Todo:
  - create a list:

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

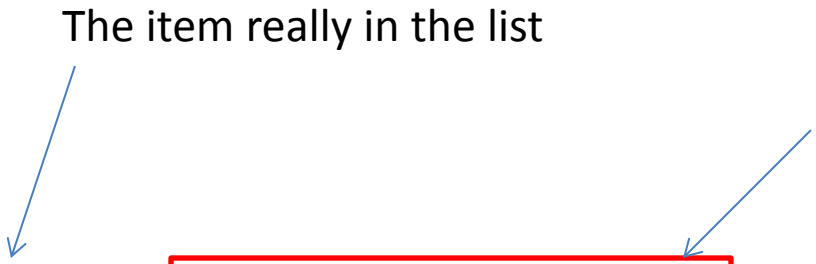
- You can

```
>>> 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




```
>>> 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
]
```

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, 3
2, 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]
```

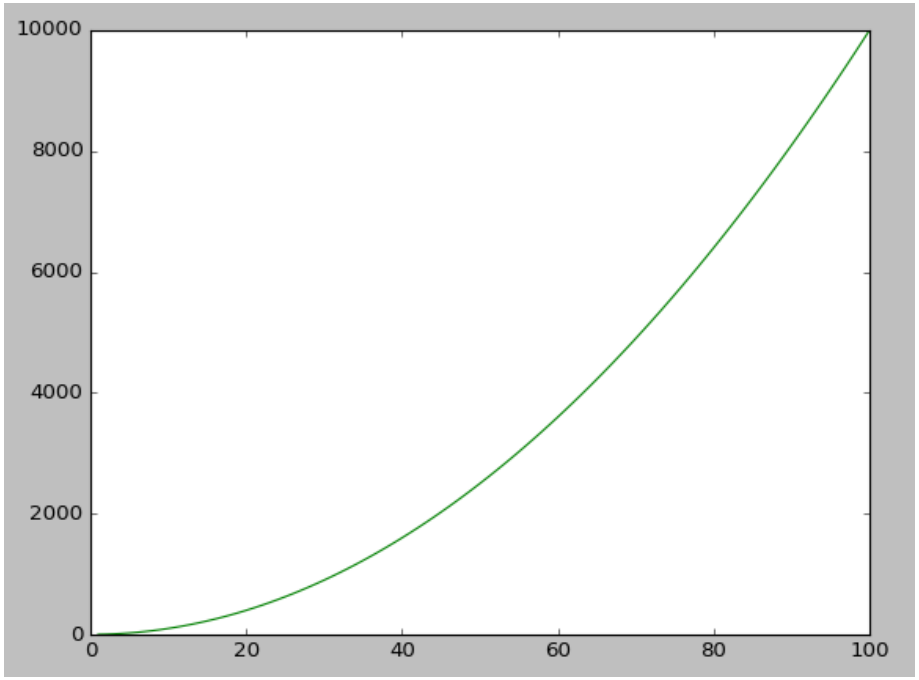
List Comprehension

- 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]
```

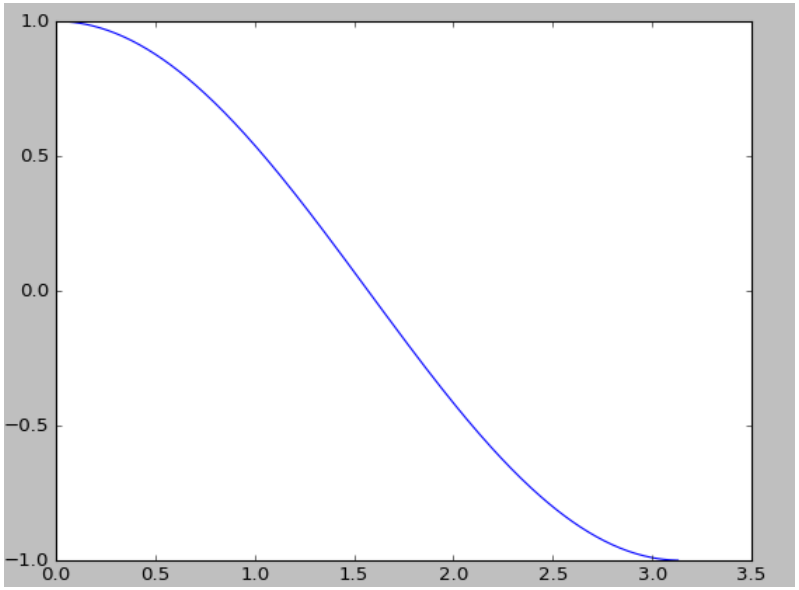
Plotting Graphs with matplot

```
>>> import matplotlib.pyplot as plt
>>>
>>> x = [i for i in range(1,101)]
>>> y = [i*i for i in range(1,101)]
>>>
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x00000001073254A8>]
>>> plt.show()
```



Plotting Graphs with matplot

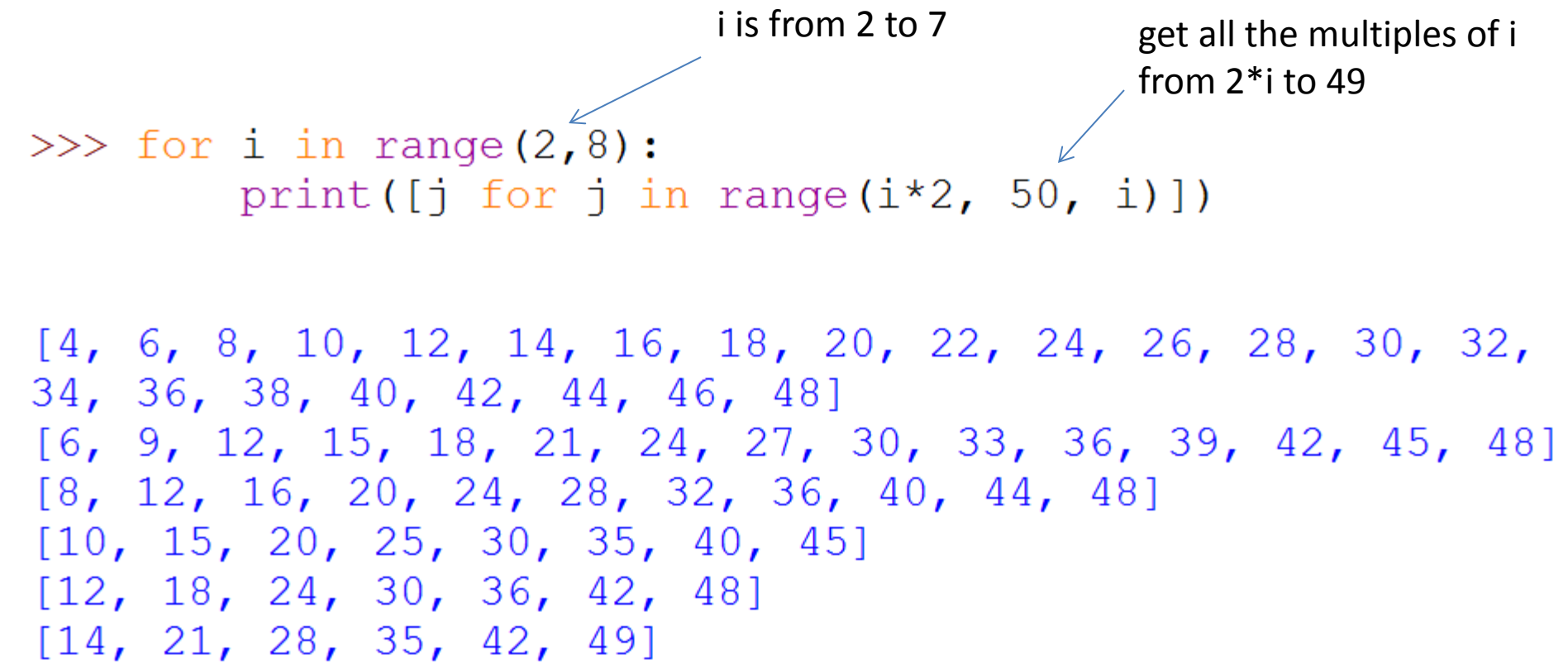
```
>>> from math import cos
>>> x = [i/100 for i in range(0,314)]
>>> y = [cos(i) for i in x]
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x00000001074EE9B0>]
>>> plt.show()
```



Challenge:
How to draw a circle?

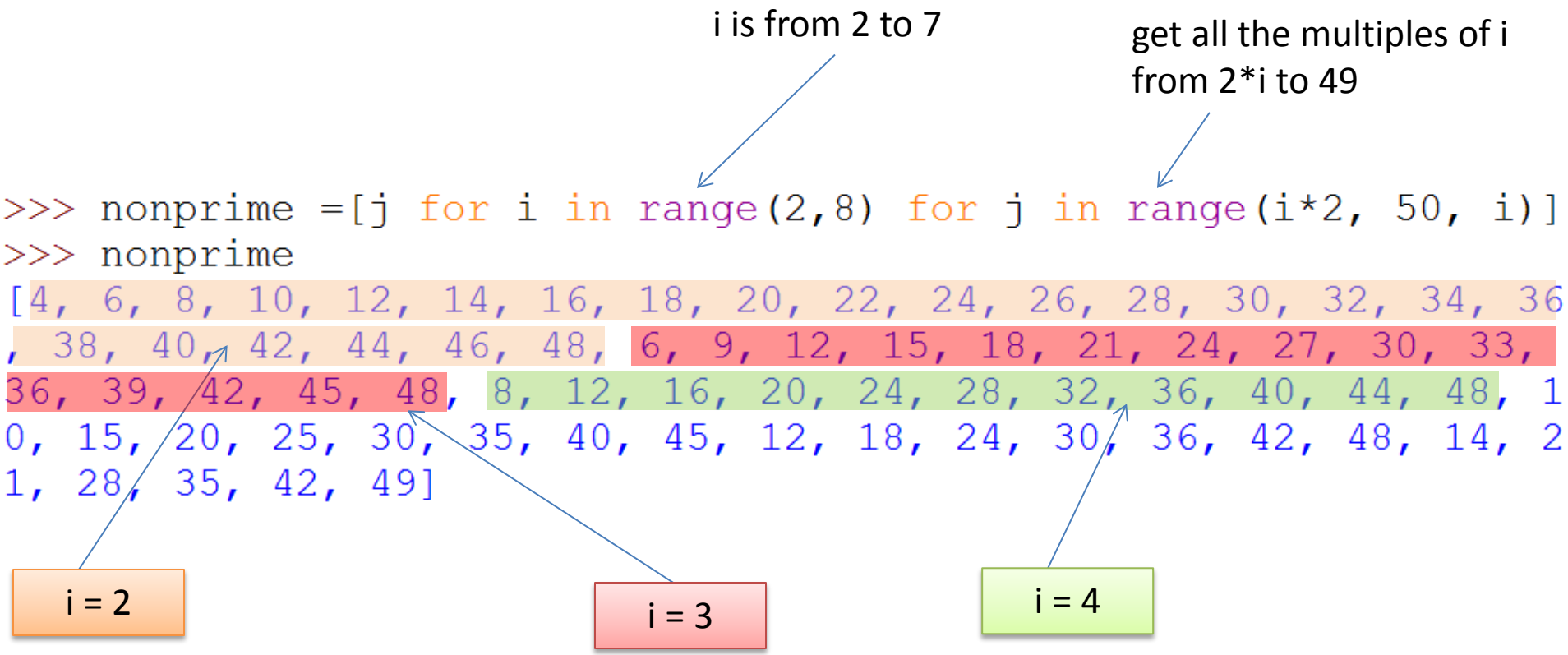
Generate Prime Numbers

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



Generate Prime Numbers

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



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

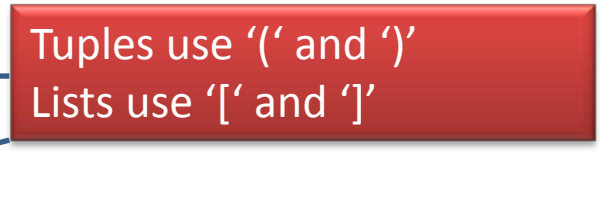
All iterables

```
>>> 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'
>>>
```

Wait

```
>>> 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

```
>>> b_tuple(0)
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    b_tuple(0)
TypeError: 'int' object is not callable
>>>
```

But then, why use Tuple? Or List?

Or when to use Tuple? When to use List?

English Grammar

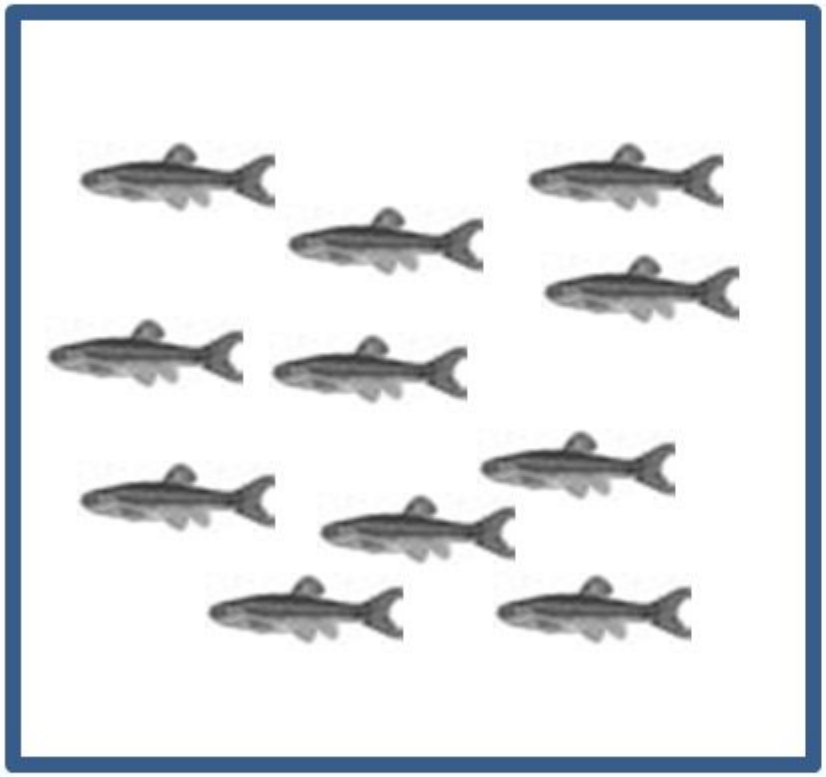
- Which sentence is grammatically correct?
  - "I have more than one fish. Therefore, I have many ***fish***"
  - "I have more than one fish. Therefore, I have many ***fishes***"

- Both of them are grammatically correct!
  - But they mean different things

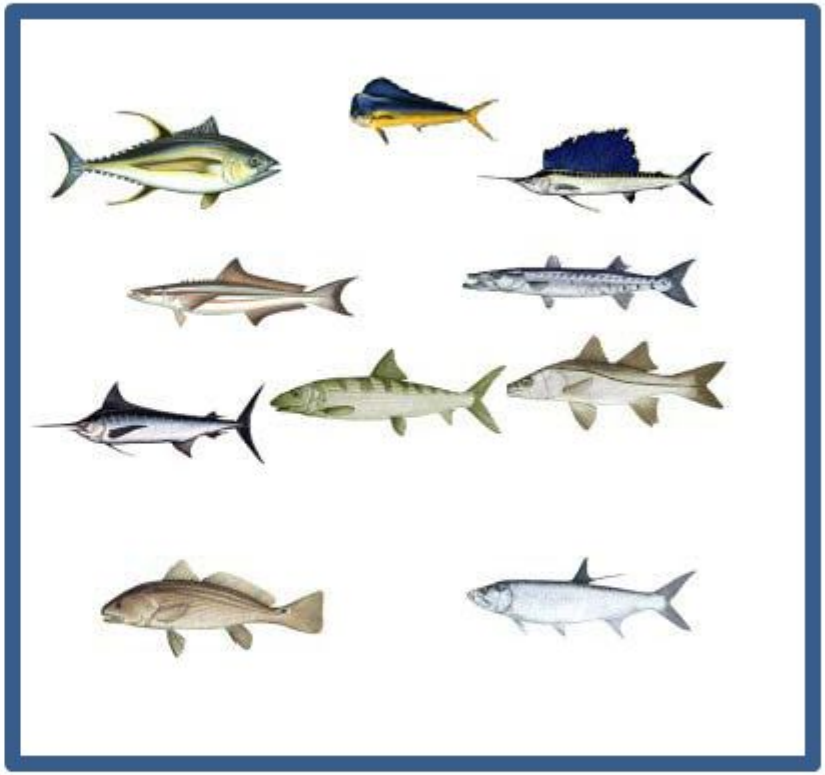
Fish vs Fishes

- 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."

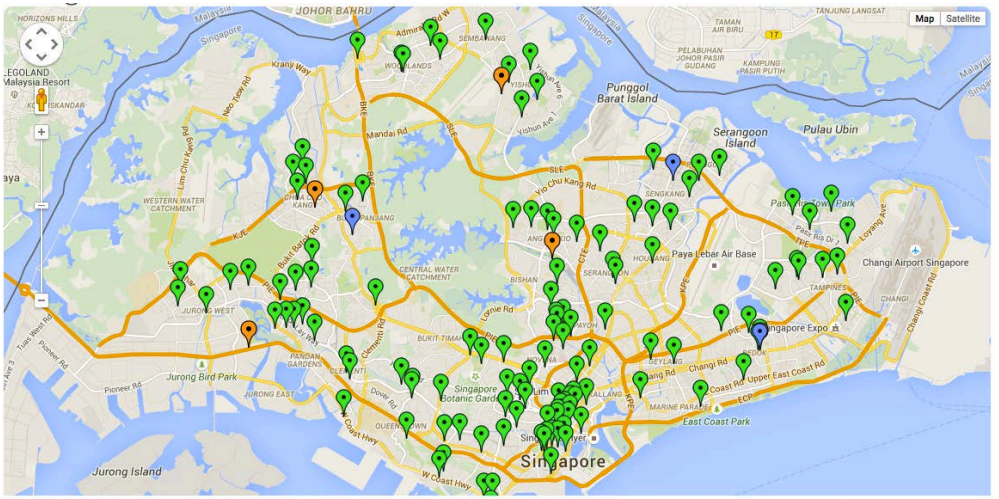
List vs Tuple, **Cultural** Reason

- 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
```

```
>>> 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)]
```

```
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
```

Just a fake function to generate the list for this demo

A list

A tuple

Challenge:
Find the nearest THREE restaurants

Instead of ALL

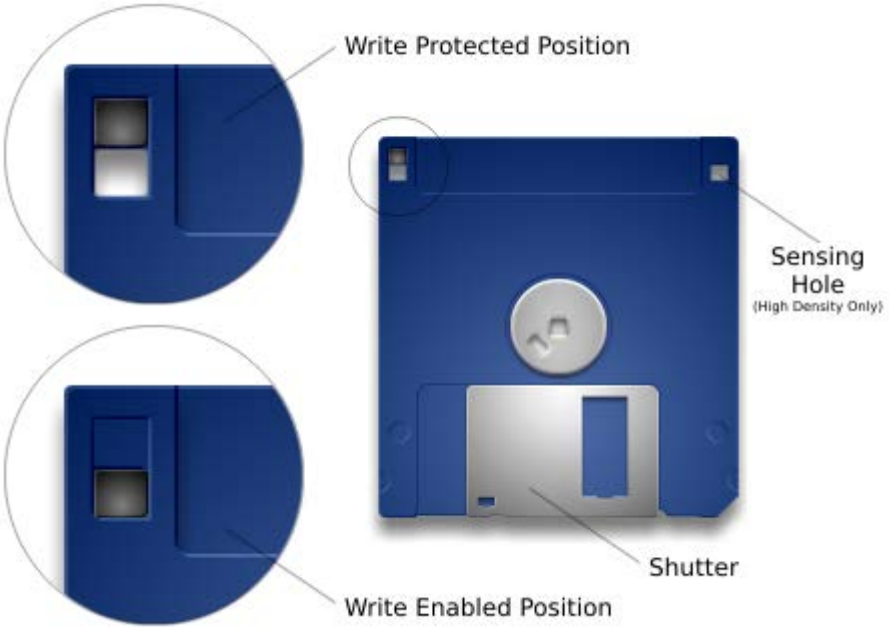
List vs Tuple, **Cultural** Reason

- 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
  - Write protected
- List can be changed within a function
  - NOT passed by value



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

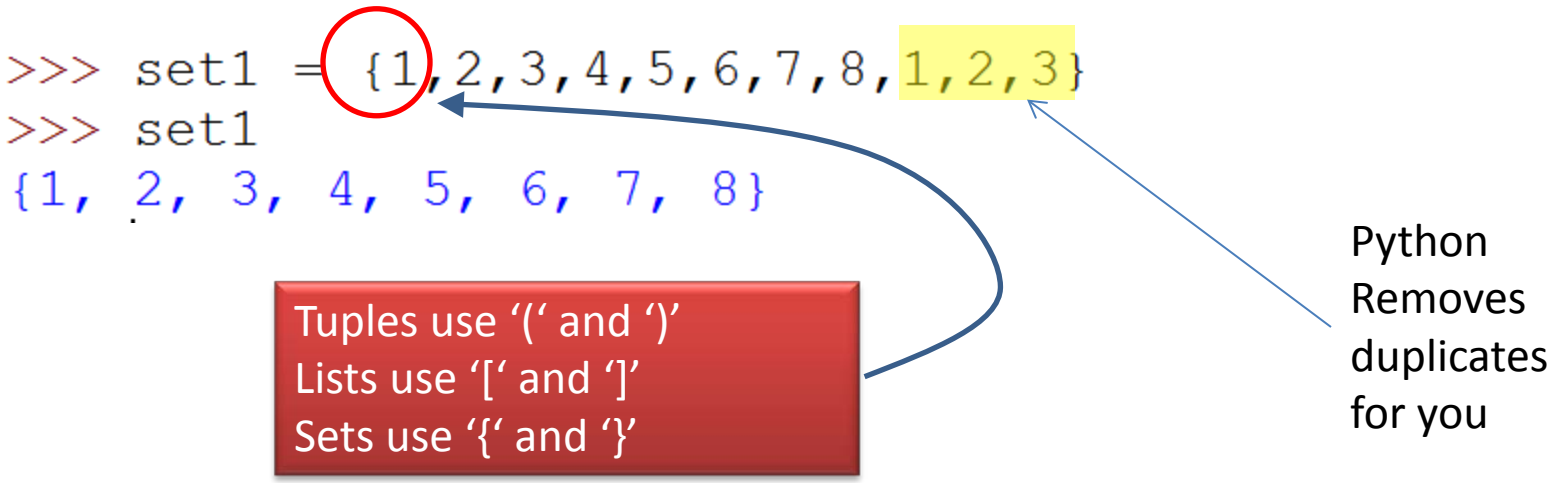
```
>>> s = 'abcde'
>>> for i in s:
        print(i)


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



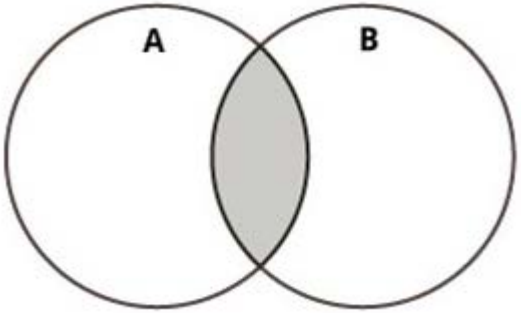
Sets

- Some operations are not available because sets are NOT indexed

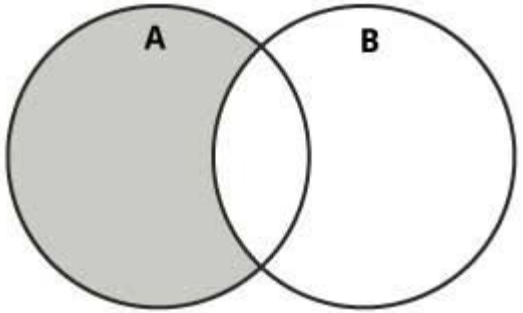
`a[i]`	return i-th element of a
`a[i:j]`	returns elements i up to j-1
`len(a)`	returns numbers of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns True if x is a part of a
`a + b`	concatenates a and b
`n * a`	creates n copies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

Set Operations

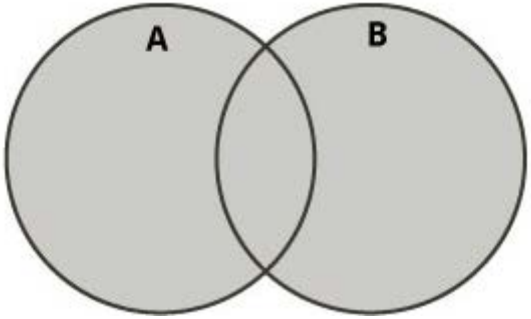
- Intersection



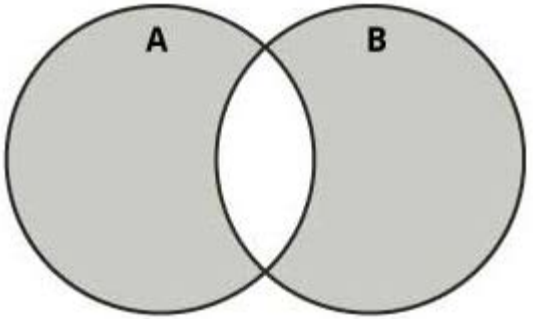
- A – B



- Union



- Symmetric Difference



Sets

- Usual set operations

```
>>> setA = {1,2,3,4}
>>> setB = {3,4,5,6}
>>> setA | setB
{1, 2, 3, 4, 5, 6}
>>> setA & setB
{3, 4}
>>> setA - setB
{1, 2}
>>> setA ^ setB
{1, 2, 5, 6}
```

Union

Intersection

A - B

(A | B) – A & B

Sets

```
>>> setA.remove(1)
>>> setA
{2, 3, 4}
>>> setA.remove(1)
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    setA.remove(1)
KeyError: 1
>>> setA.discard(1)
>>>
```

Remove like a list

But error if element missing

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

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

e•mer•i•tus (ĭ-mĕr′ĭ-təs) *adj.* Retired but retaining an honorary title: *a professor emeritus.* [Lat., p.p. of *emerēri,* to earn by service.]

em•er•y (ĕm′ə-rē, ēm′rē) *n.* A fine-grained impure corundum used for grinding and polishing. [< Gk *smuris.*]

e•met•ic (ĭ-mĕt′ĭk) *adj.* Causing vomiting. [< Gk. *emein,* to vomit.] **—e•met′ic,** *n.*

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

em•i•grate (ĕm′ĭ-grāt′) *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 (ĕm′ĭ-grā′) *n.* An emigrant, esp. a refugee from a revolution. [Fr.]

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

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

em•phat•ic (ĕm-făt′ĭk) *adj.* Expressed or performed with emphasis. [< Gk. *emphatikos.*]—**em•phat′i•cal•ly** *adv.*

em•phy•se•ma (ĕm′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.*]

em•pire (ĕm′pīr′) *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 (ĕm-pîr′i-kəl) *adj.* Also **em•pir•ic** (-pir′ĭk). **1.** Based on observation or experiment. **2.** Relying on practical experience rather than theory. [<Gk. *empeirikos,* experienced.] **—em•pir′i•cal•ly** *adv.*

em•pir•i•cism (ĕm-pîr′ĭ-sĭz′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.—**em•pir′i•cist** *n.*

em•place•ment (ĕm-plās′mənt) *n.* **1.** A prepared position for guns within a fortification. **2.** Placement. [Fr.]

em•ploy (ĕm-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.] **—em•ploy′a•ble** *adj.*

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

ă pat ā pay â care ä father ĕ pet ē be ĭ pit ī tie î pier ŏ pot ō toe ô paw, for oi noise o͝o took o͞o boot ou out th thin *th* this ŭ cut û urge yoo abuse zh vision ə about, item, edible, gallop, circus

Word

Its meaning

Word

Its meaning

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

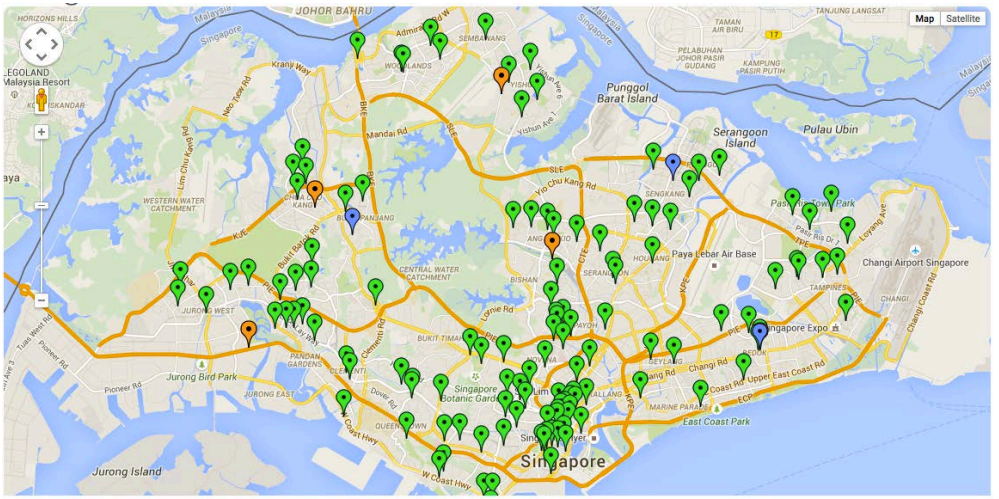
key : value pair

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

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'
```

Sequence in Python

- Indexed
  - Strings
  - Lists
  - Tuples

All iterables
Because the elements are indexed

- Non-indexed collection:
  - Sets
  - Dictionary