25/5/2020:

Tuple:

① tuple is represented using ()

② insertion order is preserved.

③ Every element in tuple is accessed using [] index (same as list, slicing)

④ Heterogenous elements are allowed

⑤ tuple is immutable data.

① i) t = (10, 20, 30)

t[0] → 10
t[-1] → 30
} Same as list.

② t = (10, 20, 30)
del t[1] → X.
del t ✓
↓ deleting entire tuple is possible.

② l = [10, 20, 30]
l[1] = 111
l.append(40) → [10, 111, 30, 40]
del l[1] → [10, 30, 40]
l.remove(30) → [10, 40]

→ In list we can,
① modify
② add new
③ delete.

→ in case of tuple,

→ insertion
→ deletion
→ modification
} → all are not possible in tuple so → tuple is "immutable".

① t = (10, 20, 30, 20, 20, 10)

t[1] = 20.
// o/p: type Error: tuple object does not support item assignment

t.append(111) → append is not there in tuple.
// o/p: Attribute Error: tuple object has not attribute append

① t.index(20) → 1

② t.count(20) → 3

③ print(t[::-1]) → (10, 20, 20, 30, 20, 10)

↳ in tuple reversing the tuple is possible the elements will remain same

④ print(t) → (10, 20, 30, 20, 20, 10)

// What is the difference b/w list & tuple?

*. List → List is mutable object

*. tuple → Tuple is immutable object.

// When & where to use tuple? list?

1→ emp id = []

↳ here we go for list because when employee leaves his id must be deleted. so tuple is not suitable.

→ list of file in directories → []

→ months in a year → ()
→ weeks in a moth → ()
→ toss coin result → ()

} → we go for tuple since we don't have change their results

// Given a list convert to tuple & vice versa

list ⇄ tuple

① L = [10, 20, 30]
t = tuple(l)
print(l) → (10, 20, 30)

② list(t) → [10, 20, 30]

→ In a tuple if any element is mutable then the content in that specific object can be modified.

eg: t = (10, 20, [30, 40, 50], 60) → tuple can be heterogenous
(indices: 10 → 0, 20 → 1, [30, 40, 50] → 2, 60 → 3; 30 → 0, 40 → 1, 50 → 2 inside the list)

t[2][1] = 111

print(t) → (10, 20, [30, 111, 50], 60)

// In tuple we can also use expressions

Eg: t = (10+3, "hello", 4>3, 5/2)

print(t) → (13, "hello", True, 2.5)

print(type(t)) → <class 'tuple'>
↓
To find the type.

// To construct a tuple with one value.

t = (10)
print(t)
print(type(t))
↳ O/P: <class 'int'>
↓
Here it will be taken as integer & not tuple

t = (10,)
print(t)
print(type(t))
↳ O/P: <class 'tuple'>
↓
→ Always to create a tuple with single element we must add a leading comma.

Reason:

In general ( ) → to group an expression Eg: (3+2) * 5
→ to represent a tuple Eg: (10, 20, 30)

So ↑ this will be grouping & so it will be taken as integer and not tuple. so leading "," is mandatory.

// in tuple ( ) is optional.

Eg: (10, 20, 30)

↳ 10, 20, 30 → it is also tuple

↓

→ It is called packing.

t = 10 → int

t = 10, → tuple.

t = "hello" → string

t = "hello", → tuple

t = () ⇒ empty tuple (X)

t = tuple() → empty tuple

l = list() →
l = [] → } → empty list.

(X) tuple unpacking:

a = 10
b = 20
c, d = a, b.
↳ input two ↳ left two } equal elements

t = 10, 20, 30
20

a, b, c = t → ✓ unpacking.
↓ ↓
10 30

// a, b = t X
only 2 elements but we have 3

a, b, c, d = t X
↓
here 4 elements but we have only 3.

} These unpacking is not possible because while unpacking the no. of elements of the tuple must be equal, (i.e) the size of the tuple must be same.

(X)

→ t = (10, 20, 30)

print (t * 2) → (10, 20, 30, 10, 20, 30)

print (t) → (10, 20, 30)

→ Here (t*2) → does not have any impact on the original tuple.

→ we use * (repition), + (concatenation) on tuple.

→ a = 5
print(a+1) → 6.
print(a) → a
→ here a+1 does not have impact on a. similarly ↵

(X) → empty tuple () (or) empty list → [] is always False

↓

Eg:

```
if []:                      # ↦ list empty
    print ("hello")
else:
    print ("hai")
```

O/P: hai

```
if ():                      # ↦ tuple empty
    print("hello")
Else:
    print ("hai")
```

O/P: hai

Dictionary → dict:

① It is represented using key-value pairs.

② { } is used to represent dictionary.

③ insertion order is not considered.

④ duplicate keys are not allowed.

⑤ duplicate values are allowed.

⑥ key must be immutable

⑦ value can be either mutable / immutable.

⑧ → employee → empId, empname

Eg: key : value

Eg: d = { 222 : 'ram', 333 : 'sam', 111 : 'ram', 444 : 'kai' }
(key: 222, 333, 111, 444; value: 'ram', 'sam', 'ram', 'kai')

① If I want to change kai to kaithik → d[444] = "kaithik"

② If I want to delete ram in 111 → del d[111]

③ If I want to add jam → d[125] = 'jam'

→ here 125 key is not already present

so a new key pair will be created : {d= 222:'ram', 333:'sam', 444:'kaithik', 125:'jam'}
and inserted any where. we don't know insertion order.

→ d = {2: 'man', 4: 'sam', 1: 'jam', 3: 'dam'}

→ dictionary iteration is done through keys.

① for (i) in d: → here i iterates through key.

    print(i) → o/p: 2
                    4
                    1
                    3

// To print value of dict.

② for i in d:

    print(d[i]) → o/p: man
                       sam
                       jam
                       dam

// To print both key & value:

③ for i in d:

    print(i, d[i]) → o/p: 2 man
                          4 sam
                          1 jam
                          3 dam

// List:

→ l = [10, 20, 30]; iterates through elements

for (i) in l:

    print(i). → o/p: 10
                     20
                     30

// tuple.

→ t = (111, 222) iterates through elements

for (i) in t:

    print(i) → o/p: 111
                    222

// dictionary can be converted to list or tuple.

→ l = tuple(d1) → (2, 4, 1, 3)

  l = list(d1) → [2, 4, 1, 3]

2/6/15

dict → key value pairs.

① d = {1: 'ram', 7: 'sam', 9: 'jam', 7: 'kar'}.

print(d[7])

↳ o/p : kar.

↳ Here we have 2 key values '7' but if we print it we will only get the latest value.

② li = list(d) → [1, 7, 9]

↳ keys (list)

③ tu = tuple(d) → (1, 7, 9)

↳ keys (tuple)

④ li = [10, 20, 30] d = dict(li) → Type Error

// print(dict(li)) → Type Error:

↳ we need the key value pairs to convert list to dict.

List of key Value pairs:

① li = [[1, "abc"], [6, "kar"], [7, "ram"]]

↳ List of lists

② li = [(1, "abc"), (6, "kar"), (7, "ram")]

↳ List of tuples

③ tu = ((1, "abc"), (6, "kar"))

↳ tuple of tuples

④ tu = ([1, "abc"], [6, "kar"])

↳ tuple of lists.

⑤ dict(li)

⑥ dict(tu)

- ① Which of the following is not valid ?

① d = { 1: 'abc', 'abc' : 1 } ✓

② d = { 'abc' : 100, 'ghi' : 200 } ✓

③ d = { True : 'ON', False : 'OFF' } ✓

④ d = { ["red", "green", "orange"] : "color",
↳ here key is list which is mutable but keys must be immutable.
["mango", "grapes"] : "fruit" } ✗
↳ It is invalid because list is mutable

⑤ d = { "apple" : ["small", "medium", "large"],
"grapes" : ["green", "black", "violet"] } ✓

→ Except 4 all are valid .

Value → mutable / immutable
key → immutable. (✗)

→

Note :

(✗) All numbers are immutable but number out of (-5 to 256) every time the object is created. For (-5 to 256) the object is created only once by the interpreter already.

// d = {231: "kau", 425: "sam", 176: "ram"}

① d.items() → both key & value returns → dict_items in the form → list of tuple

print(① li = d.items()
print(li) → dict_items([(23, "kau")(425, "sam"),
② li = list(d.items()) (176, "ram")])
print(li) → [(

② d.keys() → only keys → dict_keys → (list of keys)

① li = d.keys()
print(li) → dict_keys([231, 425, 176])

② li = tuple(d.keys())
print(li) → (231, 425, 176)

③ d.values() → only values → dict_values → (list of values)

① li = d.values()
Print(li) → dict_values(['kau', 'sam', 'ram'])

② li = list(d.values())
print(li) → ['kau', 'sam', 'ram']

// pop() → pop(key, default value)
↳ any value

① if key is present the corresponding value of the key is returned
② else default value is returned.

~~d={7:'aaa'}~~

d = {777: 'ram', 999: 'jam', 41: 'sam'}

① d.pop(999) → jam

② d.pop(125) → KeyError
↳ since 125 key is not in d it returns error.

③ d.pop(125, None) → None
→ in pop we can give the 2nd parameter, if the key is not there the second parameter will be returned.

④ d.pop(125, "NA") → NA
↓
125 not there so "NA" is printed. The second parameter can be any value.

⑤ d.pop(999, "NA") → jam
↓
is present so 'jam' is printed

⑥ d.popitem() → (111, sam)
↓
return the last item if the insertion order is preserved else returns a random value.

⑦ d.pop(125, 1) → 1
↓
not present.

// setdefault() → To create a new keyvalue pair.

① d = {1: 'abc', 3: 'def', 4: 'ghi'}

// d.setdefault(7, 'lmn') → {1: 'abc', 3: 'def', 4: 'ghi', 7: 'lmn'}
d[7] = "lmn" ⟷ 7 is new key so added

// d.setdefault(3, 'kar') → {1: 'abc', 3: 'def', 4: 'ghi', 7: 'lmn'}
3 already present so it will not be added, returns the same dict

If we give d[3] = 'kal' the 3 value will be changed but in setdefault it is not possible

3 remains same

Set: → To represent unique objects values we go for set.
↳ used for membership testing → x is present or not in set.

① insertion order is not preserved (X)

② represent using {}

③ duplicates are not allowed. (X)

④ heterogenous are allowed. (but non-recommended)
↳ MCQ (X)

⑤ Set object does not support indexing → Eg: s1[0] X

Eg: s1 = {10, 20, 30, 20, 10, 10, 30}

print(set(s1)) → {20, 10, 30}
↳ duplicates are removed
↳ insertion order not preserved.

// To add elements in set we can use add()

s1 = {20, 10, 30, 40}

s1.add(50)

s1.add(20)

// print(s1) → {20, 10, 30, 50, 40}

s2 = {60, 10, 70, 20}

s1.update(s2)

// print(s1) → {20, 10, 70, 30, 40, 60}

(X) add
↳ will add only one element

(X) update
↳ will add multiple elements.

add() & update()
↳ can be used in list too.

Set operations: (we represent set using Venn diagram)

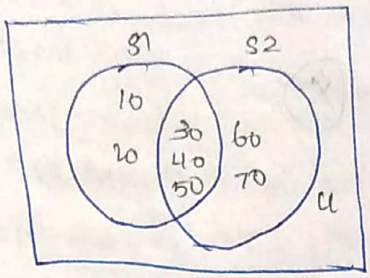
$S1 = \{10, 20, 30, 40, 50\}$

$S2 = \{30, 40, 50, 60, 70\}$

Eg: For Venn dia.

60 ⇒ MP
40 ⇒ PC
100 ⇒ MPB

Venn diagram:



① $S1 \& S2 \longrightarrow \{30, 40, 50\}$
↓
Intersection.

② $S1 \wedge S2 \longrightarrow \{10, 20, 60, 70\}$
↓
$[(S1-S2) \cup (S2-S1)]$ → Symmetric_difference.
(i.e only S1 & only S2.

③ $S1 \mid S2 \longrightarrow \{10, 20, 30, 40, 50, 60, 70\}$
↓
union

④ $S1 - S2 \longrightarrow \{10, 20\}$
↓
(only S1) Set difference

⑤ $S2 - S1 \longrightarrow \{60, 70\}$
(only S2) Set difference

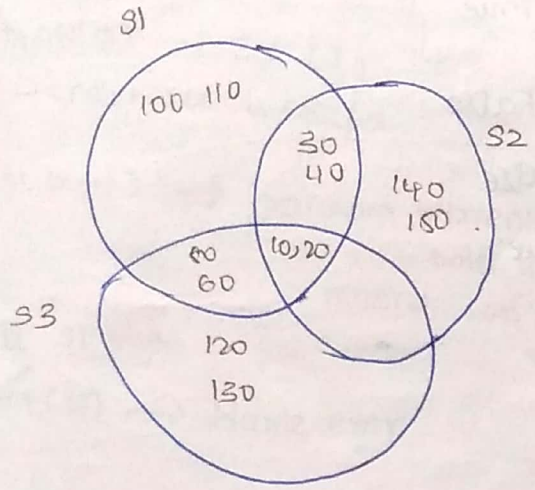
⑥ $S1 + S2 \longrightarrow$ Error (Type Error)
↓
not possible in set.

Eg:

S1 = { 10, 20, 30, 40, 50, 60, 100, 110 }

S2 = { 10, 20, 30, 40, 70, 80, 140, 150 }

S3 = { 10, 20, 50, 60, 70, 80, 120, 130 }



1) S1 & S2 & S3 →

2) (S1 & S2) - S3 →

3) S1 & S2 ∧ S3 →

4) S1 ∧ S2 ∧ S3 →

5) S1 ∧ S2 ⅋ S3 →

→ Both are possible.

S1 & S2 → S1. intersection (S2)

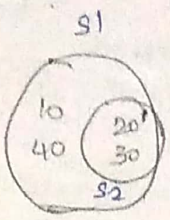
S1 - S2 → S1. difference (S2)

(S1 - S2) U (S2 - S1) → S1. symmetric_difference (S2)

S1 | S2 → S1. union (S2)

Eg: S1 = {10, 20, 30}
S2 = {40, 50}

//S1. isdisjoint (S2) → True
↓
no common elements.

② 

S1 → superset
S2 → subset of S1.

S1 = {10, 20, 30, 40}

S2 = {20, 30}

→ S1.issuperset(S2) ⇒ True

→ S2.issuperset(S1) ⇒ False

→ S1.issubset(S2) ⇒ False

→ S2.issubset(S1) ⇒ True

③ S1 = {10, 20, 30, 40, 50}

S2 = {30, 40, 50, 60, 70}

// S1.intersection(S2) → {30, 40, 50}
// print(S1) → ~~{30, 40, 50}~~ {10, 20, 30, 40, 50} → result will not be updated.
S1.intersection_update(S2)
↳ result is updated to S1.

Print(S1) → {30, 40, 50}
↓
only the elements updated are printed

→ similarly we can use,

✓ ① symmetric_difference_update

✓ ② difference_update()
↓
here result is updated to set.

③ symmetric_difference()

④ difference()
↓
here result will not be updated.

//discard(): → to remove a element from set.

① S1 = {10, 20, 30, 40}

//S1.discard(20) → 20 is discarded

print(S1) → {10, 30, 40}

② S1.clear()

print(S1) → set().

↳ entire set is cleared.

} → applicable in dictionary too.

③ S1.pop() → random elements is removed since insertion order is not preserved but pop() generally removes last element.

④ del S1

print(S1) → Name Error.

⑤ S1.symmetric_difference(S2)

↳ (S1 - S2) U (S2 - S1)

result will not be updated

⑥ S1.symmetric_difference_update(S2)

↳ S1 = (S1 - S2) U (S2 - S1)

Result will be updated.

empty representation:

① k = { } → dictionary

print(type(k)) → <type 'dict'>

② S = set() ⇒ empty set

Frozen set: → we cannot modify the content.

s1 = {10, 20, 5, 20, 5, 5, 10, 7}.

fs = frozenset(s1)

s1.add(222) → ✓

fs.add(222) → attribute error.

∵ It is frozen set

tuple	frozenset
duplicates are allowed	duplicates are not allowed

Month: Jan, feb, ..., Dec → go for frozenset.

To remove elements

discard()	remove()
→ If an element is present then it is removed otherwise ignored.	If an element is present removed else key error.
Eg: s1 = {10, 20, 30}	s1 = {10, 20, 30}
① s1.discard(10) ✓	s1.remove(10) ✓
s1.discard(111)	s1.remove(111)
↓ ignored	↳ key Error. 111 is not in the set

→ What is the difference b/w set and frozenset

Set → mutable

Frozen set → immutable

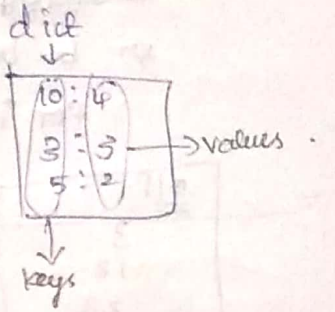
// WAP to find the occurance count of every element in a list.

```
l = [10, 3, 5, 3, 3, 5, 10, 10, 10]
```

10 → 4
3 → 3
5 → 2

prgm:

```
s = set(l)
d = {}
for i in s:
    if i not in d.keys():
        d[i] = 1
    else:
        d[i] += 1
```



Datatype → (bytes) ⇒ 0 to 255 [0 - 256] ↳ exclusive.

→ The content in the bytes cannot be modified.

→ Bytes → immutable datatype.

```
l = [3, 13, 23, 123]
b = bytes(l)
for i in b:
    print(b)
```

→ o/p: 3
13
23
123

Where we use bytes?
→ we use bytes datatype in image related data or operations, since the pixel range is 0 to 255

→ it can hold values only from 0 to 255

Eg:

```
l = [3, 330]
b = bytes(l) → Error
```

(330 → out of range)

⑤ Bytearray datatype:

↳ here modification can be done.

```
l = [3, 13, 23, 123]
b = bytearray(l)
for i in b:
    print(i)
```

O/P

3
13
23
123

b → [3, 13, 23, 111]

```
// b[3] = 111
print(b)
```

O/P:

3
13
23
111 → modification is allowed

Note:

→ In bytes datatype
↳ values cannot be modified

→ In bytearray datatype
↳ values can be modified.

Bytes → String:

```
// s = "abcd"
b = bytes(s)
```

→ String

↳ O/P: Error: Encoding Error,

```
s = b"abcd"
b = bytes(s)
for i in b:
    print(i)
```

→ binary/bytes → Binary string

ascii values will be considered.

O/P: 97
98
99
100