Other Structured Types: Sets and Tuples

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Recap

- Variables point to objects of different types
- Objects can be of scalar types: int, float, boolean
- Objects can be structured/ compound types; built-in are: lists, strings, sets, tuples, dictionaries
- In structured types, you can not only perform operations on the whole object, but can also extract items from it
- So far we have discussed lists and strings
- Now we will discuss the remaining ones: tuples, sets, dictionaries
- Dictionaries in particular are very commonly used, and like strings, are a strength of python

Recap – Lists

- Lists are a list of items in brackets, eg. [1, 4, 9, "str", 5.0, 4]
- Lists are mutable, can change the items like L[index] = val
- Can slice a list to get a sublist - from start or from end of list
- Joining or repeating lists by operations: + , *
- Functions with list as parameter: len(), sum()
- Presence/absence of item by ops: in, not in
- Ops on a list: append(), insert(), extend(), remove(), pop(), index(), reverse(), count(), copy(), sort()
- Can easily loop over list items - item by item, or using index
- Lists can be nested - i.e. list items are themselves lists
- List comprehension a compact way to create lists from lists

Recap – Strings

- Strings are like "Hello hi" or 'Hi Hello' ; can loop over chars in str
- They are immutable, cannot change any item of a string
- Can slice a string to get substrings - from start or from end
- Functions: len(), in, not in, + , *
- Can split strings into a list of items using split()
- Can join a list of strings to form one using join()
- String operations (return a new string): lower(), upper(), replace(), count(), find(), isdigit(), …

Creation using list() and str()

- We can also create a list using the constructor function list()
- It takes one parameter, and all elements of it become elts of list

  list("str") is ['s', 't', 'r']

  list((1, 3, 5)) is [1, 3, 5] # list(1,3,5) is error - one arg allowed

  list() returns a null list []

- String also has a constructor: str(arg) : takes whatever is the argument and returns as a string

  str(54) returns '54', str([1,4]) returns '[1, 4]', …

- Such constructors are much more useful in converting objects of one structured type to another

Format Strings

- Format strings help format the output. So far to print, we have been using

```
print(str, var, str, var, …)  #Strings and vars separated by ,
print("The age of: ", name, " is ", age, "on: ", date)
```

- Can become hard to read, and map output to the print stmt.
- Better methods needed to mix commentary-strings and values being printed - f strings are commonly used now.

```
print(f'The age of {name} is {age} on {date} ')
```

- Format strings - prefix str with f with expressions/vars embedded in it between { } - their values replace them at processing time
- There are other older methods also to do formatting - using %, or the function string.format()

Sets

Sets

- Sets are like the mathematical concept
- Used to store multiple items - items can be of different types
- A set is unordered, unindexed, and without duplicates
- Sets are written with curly brackets, Examples:-

```
Int_set = {1, 3, 5, 8}
colors  = {"red", "blue", "yellow"}
```

- Set itself is mutable - can change a set by deleting, adding,...
- However, set values are immutable - so can have strings, int, float, but not lists as set items.
- Example: s = {1,2, [1,2] } will give the following error: TypeError: unhashable type: 'list'

Sets – Unordered but Iterable

- Sets don't have order - cannot refer to nth item; internally also python may save them in its own order, e.g.

  s = {'q', 'u', 'u', 'x'}, value of s is: {'x', 'u', 'q'}
- However we can loop over a set - it will go over all items (in some order), but cannot loop over them by index, as in list
-
- len() function is also defined - returns the number of items
- in and not in operations also work as with list/ str

Set Operations – Example

Can loop through set items:

for item in <set>:

<loop-body>

Cannot loop using index - as items are not ordered as in list

Can check presence/absence of an item in a set by *in* and *not in*

```
print(“hello” in s) # True
print(“Hello” in s) # False
print(1 not in s) # False
print(“Hello” not in s) # True
```

```
s = {1,2,"hello", "world",1,2}
print(s) # {1, 2, 'hello', 'world'}
print(len(s)) # 4
print(type(s)) # <class 'set'>
for element in s:
    print(element)

Output:
    1
    2
    hello
    world
```

Operations to Modify a Set

Following operations are allowed on a set s (modifies the set s)

s.add(item) # will add item to s, if does not already exist

s.remove(item) # will remove item if it exists, error otherwise

s.discard(item) # no error if item does not exist

s.clear() # clears the set

s.update(s2) # s2 is a set, list, tuple: adds elts of s2 to s - duplicates dropped, i.e. does a union operation

del s # Completely deletes the set s

Quiz – Single Correct

What would be the output of the code given at the right?

a.) [1,2,3,4,4,5,6]

b.) {1,2,3,4,5,6}

c.) {1,2,3, [4,5,6]}

d.) [1,2,3,4,4, 5,6]

```
s = {1, 2, 3, 4}
L = [4,5,6]
s.update(L)
print(s)
```

Quiz – Single Correct

What would be the output of the code given at the right?

a.) [1,2,3,4,4,5,6]

b.) **{1,2,3,4,5,6}**

c.) {1,2,3, [4,5,6]}

d.) Error

```
s = {1, 2, 3, 4}
L = [4,5,6]
s.update(L)
print(s)
```

Explanation: s.update(s2) adds elts of s2 to s, no duplicates

Operations on Sets

In these operations the original sets remain unchanged and a new set is returned.

- **s1.union(s2)** # returns the union of s1 and s2
- Can also be done by s1 | s2
- Can have union of multiple sets, s1.union(s2, s3); s1|s2|s3

- **s1.intersection(s2)** # the intersection of s1 and s2
- Can also be done by: s1 & s2
- Can have intersection of multiple sets.

- **s1.difference(s2)** # items in s1 which are not in s2
- Can also be done by: s1 - s2

Operations on Sets

- **s1.symmetric_difference(s2)** # items in s1 or s2 but not both
- Can also be done by: s1 ^ s2

- **s1.isdisjoint(s2)** # True if s1 and s2 are disjoint
- **s1.issubset(s2)** # True if s1 is subset of s2
- **s1.issuperset(s2)** # is s1 a superset of s2

- Relational operators (<, <=, >, >=, ==, !=) also defined: s1 < s2 if s1 is a subset of s2

Quiz – Single Correct

What would be the output of the code given at the right?

a.) {1, 4}

b.) {1,2,3,2,3,4}

c.) {1,2,3,4}

d.) {2,3}

```
a = {1, 2, 3}
b = {2, 3, 4}

res = (a-b)|(b-a)
print(res)
```

Quiz – Single Correct

What would be the output of the code given at the right?

```
a = {1, 2, 3}
b = {2, 3, 4}

res = (a-b)|(b-a)
print(res)
```

a.) {1, 4}

b.) {1,2,3,2,3,4}

c.) {1,2,3,4}

d.) {2,3}

Explanation : (a-b)|(b-a) = {1} union {4} = {1,4}

Set Examples

Determine if all vowels are present or not in a string

Note: For this problem uppercase/lowercase letters are still just vowels (or not)

Test case 1:
Input string: "CSE101 : Introduction to programming "
Output:
All vowels present

Test case 2:
Input string: "CS101 : Introduction to programming "
Output:
Not all vowels present
Missing : { 'e'}

```
s = input("Enter a string\n")

l = s.lower() #all lowercase letters
vowel = {'a', 'e', 'i', 'o', 'u'}

l_set = set(l) # removes duplicates

if len(l_set.intersection(vowel))==5:
  print("All vowels present")

else:
  miss = vowel.difference(l_set)
  print("Not all vowels present")
  print("Missing :",miss)
```

Set Examples

Determine if a string is a Pangram or not. A Pangram is a string that contains every letter in the English alphabet.

Test case 1:
Input string: "The quick brown fox jumps over the lazy dog"
Output:
Pangram

Test case 2:
Input string: "The quick fox jumps over the lazy dog"
Output:
Not a pangram

```
s = input("Enter a string")
l = s.lower()
ls = set(l) # Remove duplicates

# Remove digits or special chars
chars = [ch for ch in ls if ch>='a'and
ch<='z']

if len(chars)==26:
    print('Pangram')
else:
    print("Not a pangram")
```

Set Comprehensions

- Are just like list comprehensions - can be used to create new sets
- Original example we had:

    S = {x: x=n*(n+1) where 0<n<6} # from CBSE book

    Ans: S = {2, 6, 12, 20, 30}

- Set comprehension for this:

    S = {n*(n+1) for n in range(1,6)}

- Set comprehension :

    newset = { expr(elt) for elt in list/set if condition }

- Sets can also be created by constructor: set(list/tuple)

Frozen Sets

- Frozensets are like sets but are immutable - i.e. cannot be changed
- Can convert a set (or a list, tuple) into a frozenset by

  frozenset(s)

- Can perform all set operations, except add, delete, …
- With frozensets, you can define a set of frozensets (but cannot have a set of sets)
- Wherever immutable objects are required, frozensets can be used, but not sets.

Quiz

Which of the following statements is/are true about sets in Python?

A. Sets are ordered collections of unique elements.

B. Sets are mutable.

C. Sets can contain elements of different types.

D. Sets can be indexed using integers.

Quiz(Solution)

Which of the following statements is/are true about sets in Python?

A. Sets are ordered collections of unique elements.

B. Sets are mutable.

C. Sets can contain elements of different types.

D. Sets can be indexed using integers.

Options B, C are correct

Tuples

Tuples

- Tuples are used to store multiple items in a single variable; it is an ordered collection of items (of same or different types)
- Tuples are immutable (unlike lists)
- Tuples are immutable (but its elements may be mutable.)
- Tuples are written with round brackets, Examples

```
Xy-coord = (5.0, 3.1)
properties = ("Toyota", "red", 2.0, 2021)
colors  = ("red", "blue", "yellow")
t =  (1,2,1,4)  # Duplicates allowed
x = (1, 2, [5,6,7],8) # Tuple with list as an element.
num = ( (1,2), (3,4), 5) # Nesting of tuples
```

Accessing Tuples

- Just like in lists - can access an item by indexing, can access a range of items, -ve indexes, …

```
colors  = ("red", "blue", "yellow")
colors[1], colors[-1], colors[:1], colors[1:2]...
```

- When a single item in a tuple, it has to be: (item,). This tells that it is a tuple (so, tuple ops can be performed)

```
singleton_tuple = (74,) # (74,) is a tuple but (74) is not.
```

- Check if an item exists - *in* or *not in* operation like lists/strings

Operations on Tuples

- Concatenate tuples by + : returns a new tuple

```
tup1 + tup2 # Concatenation
```

- Replicate tuples by * : returns a new tuple

```
tup1 * 4 # Replication
```

- Like in a list, can unpack elements and assign to vars .

```
v1, v2, v3 = (elt1, elt2, elt3)
```

- t.count(<item>) # number of times item occurs
- t.index(<item>) # returns the index of item
-

Note: * is meaningless in sets - not provided; + is also not provide

Looping through Tuples

As in lists - either of these

- Looping over elements in a tuple

```
for item in <tuple>:
    Loop-body
```

- Looping using indices

```
for index in range(len(<tuple>)):
    Use <tuple>[index]
```

Quiz – Single Correct

What would be the output of the code given?

a.) Error

b.) 29.001

c.) 28

d.) 29

```
tup = (True, 0.8, True, False,
11, 7, 7.2, 0.001, True)

res = 0

for i in tup:

    res = res + int(i)

print(res)
```

Quiz – Single Correct

What would be the output of the code given?

```
tup = (True, 0.8, True, False,
11, 7, 7.2, 0.001, True)

res = 0

for i in tup:

    res = res + int(i)

print(res)
```

a.) Error
b.) 29.001
c.) 28
d.) 29

Explanation :
= 1 (for True)+0 (for 0.8)+1(for True)+0(for False)+11+7+7 (for 7.2)+0(for 0.001)+1(for True)
= 28

Example

Adding elements to a Tuple using the singleton tuple.

```
T1 = (10, 20, 30, 40)
L1 = [60, 70, 80]
for item in L1:
    T1 = T1 + (item,)
print(T1)
```

```
Output : (10, 20, 30, 40, 60, 70, 80)
```

Example

Given a tuple of lists. Sort lists within the tuple.

Input:

([4,2,1], [5,3,7], [6,2,1,8], [10,9])

Output:

([1, 2, 4], [3, 5, 7], [1, 2, 6, 8], [9, 10])

```
tup = ([4,2,1], [5,3,7],
[6,2,1,8],  [10,9])

res = [sorted(l) for l in tup]
res = tuple(res)

print(res)
```

Quiz – Single Correct

Q) What is the output of the following program

```
L1 = [11,2,3,4,5]
L2 = [20,7,8,2,4]
s = 9
res=[(s-L1[i],L1[i]) for i in range(len(L1)) if (s-L1[i])==L2[i]]
```

a) [(11,20),(5,4)]

b) [(11,20),(2,7),(5,4)]

c) [(7,2),(4,5)]

d) [(11,20),(2,7)]

Quiz – Single Correct

Q) What is the output of the following program

```
L1 = [11,2,3,4,5]
L2 = [20,7,8,2,4]
s = 9
res=[(s-L1[i],L1[i]) for i in range(len(L1)) if (s-L1[i])==L2[i]]
```

a) [(11,20),(5,4)]

b) [(11,20),(2,7),(5,4)]

c) **[(7, 2), (4, 5)]**

d) [(11,20),(2,7)]

Explanation : The code is finding pairs which total to a given sum and occur at the same index in the two lists(One element from each list).

Sorting a Tuple

- sorted() : To sort a tuple, use the sorted function.
- Returns a list instead of tuple.
- Use tuple() to convert it to a tuple.

```
t = (1,4,3,2)
a = tuple(sorted(t)) # a = (1,2,3,4)
d = tuple(sorted(t, reverse = True)) # d = (4,3,2,1)
```

Summary – Sets

- Sets - like math concept - unordered collection without duplicates within {}
- Sets are mutable, but their elements must be immutable
- Can loop over sets, apply len(), in, non in ops
- Ops to modify a set: add(), remove(), update(), ...
- Ops on sets: union (|), intersection (&), difference (-)
- Can check isdisjoint(), issubset(), issuperset()
- …

Summary – Tuples

- Tuples are a collection of ordered items in () - single item must have the , separator
- Is immutable; so, can have set of tuples
- Can access an item by index, loop over by item or index
- Ops: concat (+), replicate (*) allowed
- Other ops like count(), index(), etc