```
s="this is the String. String in the python are immutable."
print(s.capitalize())
```

```
This is the string. string in the python are immutable.
```

```
s="this is the String. String in the python are immutable."
print(s.count('t'))
```

```
7
```

```
s="this is the String. String in the python are immutable."
print(s.count('tr'))
```

```
2
```

```
name='ram'
age=22
s=f"this is{name}. my age is {age}"
print(s);
```

```
this isram. my age is 22
```

```
name='ram'
age=22
s="this is{}. my age is {}".format(name,age)
print(s);
```

```
this isram. my age is 22
```

```
# list
x=[1,2,3,4,5]
print(x,type(x))
```

```
[1, 2, 3, 4, 5] <class 'list'>
```

```
# order Collection
# support any type of the data structure
x=[1,1.5,'ram','1',5j,(1,2,3),{1,2,3},{'name':"ram"},[1,2,3],12]
print(x)
```

```
[1, 1.5, 'ram', '1', 5j, (1, 2, 3), {1, 2, 3}, {'name': 'ram'}, [1, 2, 3], 12]
```

```
fruits=['apple', 'banana']
fruits[1]
```

```
'banana'
```

```
x=[1,2,[3,4,5,6],7,8,9]# nested loop to find the value of 4
x[2][1]
```

```
4
```

```
# list is the mutable object can be changed once defined
fruits=['apple','banana','orange']
fruits[0]='ram'
print(fruits)
```

```
['ram', 'banana', 'orange']
```

```
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return bool(key in self).
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, index, /)
 |      Return self[index].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |    mul  (self, value, /)
```

```
fruits=['apple','banana','orange']# using the apprnd command we can add the new list object
fruits.append("graphs")
print(fruits)
```

```
['apple', 'banana', 'orange', 'graphs']
```

```
fruits=['apple','banana','orange']# using the append command we can add the new list object =>list is added  as it is
fruits.append(['apple','banana','orange'])
print(fruits)
```

```
['apple', 'banana', 'orange', ['apple', 'banana', 'orange']]
```

```
fruits=['apple','banana','orange']# using the extend command we can add the new list object by seperating the each object of the list
fruits.extend(['apple','banana','orange'])
print(fruits)
```

```
['apple', 'banana', 'orange', 'apple', 'banana', 'orange']
```

```
fruits=['apple','banana','orange']#insert in any index using the insert
fruits.insert(1,'graphs')
print(fruits)
```

```
['apple', 'graphs', 'banana', 'orange']
```

```
fruits=['apple','banana','orange']# used to pop out the index value
print(fruits.pop(2))
print(fruits)
```

```
orange
['apple', 'banana']
```

```
# Tuple
#supports any type of the datastructure
#orderdd collection
x=(1,2.2,'abi',)
print(x,type(x))
```

```
(1, 2.2, 'abi') <class 'tuple'>
```

```
# tuple is the immutable datatyoe means its value cannot be changed once written
help(tuple)
```

```
Help on class tuple in module builtins:

class tuple(object)
 |  tuple(iterable=(), /)
 |
 |  Built-in immutable sequence.
 |
 |  If no argument is given, the constructor returns an empty tuple.
 |  If iterable is specified the tuple is initialized from iterable's items.
 |
 |  If the argument is a tuple, the return value is the same object.
 |
 |  Built-in subclasses:
 |      asyncgen_hooks
 |      UnraisableHookArgs
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return bool(key in self).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
```

```
x=(1,2,3,4,1,3,1)# used to count the value of attributes
print(x.count(1))
```

```
3
```

```
x=(1,2,3,4,1,3,1)# early found index is given
print(x.index(1))
```

```
0
```

```
#dictionary
person={
    'id':1,
    'name':"abishek",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person,type(person))
```

```
{'id': 1, 'name': 'abishek', 'age': 22, 'salary': 121323, 'contact': 1212323} <class 'dict'>
```

```
person={
    'id':1,
    'name':"abishek",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person['name'])
```

```
abishek
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
person['email']='abi@gmail.com'# to add the email do this in the dictionary
print(person)
```

```
{'id': 1, 'name': 'ram', 'age': 22, 'salary': 121323, 'contact': 1212323, 'email': 'abi@gmail.com'}
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
del person['age']# to delete from the dictionary
print(person)
```

```
{'id': 1, 'name': 'ram', 'salary': 121323, 'contact': 1212323}
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person.get('email'))# this is the error handeling technique => here is no email so the email is handeled carefully with none message
print('hello')
```

```
None
hello
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person.get('age'))# to get the age from the dictionary => get()
```

```
22
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person.keys())# to find the name of the keys in the dictionary
```

```
dict_keys(['id', 'name', 'age', 'salary', 'contact'])
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person.values())# to find the values of the keys we used this syntax in the dictionary
```

```
dict_values([1, 'ram', 22, 121323, 1212323])
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
print(person.items())# to find the keys and values in the single call we use => items() syntax
```

```
dict_items([('id', 1), ('name', 'ram'), ('age', 22), ('salary', 121323), ('contact', 1212323)])
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
person.pop('name')  # to delete from the dictionary
print(person)
```

```
{'id': 1, 'age': 22, 'salary': 121323, 'contact': 1212323}
```

```
person={
    'id':1,
    'name':"ram",
    'age':22,
    'salary':121323,
'contact':1212323
}
person.update({"email":'abi@gmail.com', 'college':'broadway'})# to add the email do this in the dictionary
print(person)
```

```
{'id': 1, 'name': 'ram', 'age': 22, 'salary': 121323, 'contact': 1212323, 'email': 'abi@gmail.com', 'college': 'broadway'}
```

```
# tuple, int float, complex(number datatypes ), string=> immutable
#list, dictionary,set=>mutable
```

```
#practicing the set
#set dattype starts
s={1,2,3,4}
print(s,type(s))
```

```
{1, 2, 3, 4} <class 'set'>
```

```
# unordered collection=> we may not find the ordered output as input in the set
s={'ram',1,4,5,(1,2,3),3}
print(s)
```

```
{1, 3, 4, 5, (1, 2, 3), 'ram'}
```

```
# unique ordered collection=> here in the output  has declined the repitation and makes it as single ="ram"
s={'ram',1,4,5,(1,2,3),3,'ram',1}
print(s)
```

```
{1, 3, 4, 5, (1, 2, 3), 'ram'}
```

```
# in set only mutable datastructure can be placed
```

```
tea={'ram','hari','gita'}# only once the repeted value arre come # union of set
coffee={'ram','abi','hary'}
tea_or_coffee=tea | coffee
print(tea_or_coffee)
```

```
{'gita', 'hari', 'hary', 'abi', 'ram'}
```

```
tea={'ram','hari','gita'}# only once the repeted value arre come # union of set
coffee={'ram','abi','hary'}
tea_or_coffee=tea.union(coffee)#union of the sets
print(tea_or_coffee)
```

```
{'gita', 'hari', 'hary', 'abi', 'ram'}
```

```
tea={'ram','hari','gita'}#
coffee={'ram','abi','hary'}
tea_or_coffee=tea &coffee#intersection of the sets
print(tea_or_coffee)
```

```
{'ram'}
```

```
tea={'ram','hari','gita'}# intersection
coffee={'ram','abi','hary'}
tea_or_coffee=tea.intersection(coffee)#intersection
print(tea_or_coffee)
```

```
{'ram'}
```

```
tea={'ram','hari','gita'}# difference => in the set
coffee={'ram','abi','hary'}
tea_or_coffee=tea - coffee
print(tea_or_coffee)
```

```
{'gita', 'hari'}
```

```
tea={'ram','hari','gita'}# difference in the set
coffee={'ram','abi','hary'}
tea_or_coffee=tea.difference(coffee)#difference
print(tea_or_coffee)
```

```
{'gita', 'hari'}
```

```
help(set)
```

```
Help on class set in module builtins:

class set(object)
 |  set(iterable=(), /)
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(self, object, /)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __ixor__(self, value, /)
 |      Return self^=value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __or__(self, value, /)
```

```
s={'ram',1,4,5,(1,2,3),3,'ram',1}# to add the 33 in the set
s.add('33')
print(s)
```

```
{1, '33', 3, 4, 5, (1, 2, 3), 'ram'}
```

```
s={'ram',1,4,5,(1,2,3),3,'ram',1}# to remove the value from the set
s.discard(1)
print(s)
```

```
{3, 4, 5, (1, 2, 3), 'ram'}
```

```
len([1,2,3,4,5])# to find the length of the list
```

```
5
```

```
max([1,2,3,4,5])# to find the max vslue of the list
```

```
5
```

```
sum([1,2,3,4,5])# to find the sum of the list
```

```
15
```

```
sorted([1,2,3,4,5])# sorting in the ascending order
```

```
[1, 2, 3, 4, 5]
```

```
min([1,2,3,4,5])
```

```
1
```

```
sorted([1,2,3,4,5],reverse=True)
```

```
[5, 4, 3, 2, 1]
```

Start coding or generate with AI.

Start coding or generate with AI.

===

📘 Day 2 Python Notes: Collections in Python (Full Guide)

===

===

🔹 LIST (Ordered | Mutable | Allows Duplicates)

===

✅ Creating a List

my_list = [10, 20, 30, 20, "hello", True]

✅ Accessing Elements

first_item = my_list[0] # First item last_item = my_list[-1] # Last item

✅ Updating Elements

my_list[1] = 25 # Change 20 to 25

✅ Adding Elements

my_list.append(40) # Adds 40 to the end my_list.insert(2, 100) # Inserts 100 at index 2

✅ Removing Elements

my_list.remove(20) # Removes first occurrence of 20 removed = my_list.pop() # Removes last item my_list.clear() # Clears entire list

✅ List Slicing

my_list[1:4] # Elements from index 1 to 3 my_list[:3] # Elements from start to index 2 my_list[::2] # Every second element

✅ Useful List Methods

- append(x) → Add item at end

- insert(i, x) → Insert at index i

- remove(x) → Remove first occurrence

- pop([i]) → Remove item at index i (last by default)

- clear() → Remove all items

- index(x) → First index of x

- count(x) → Count of x

- sort() → Sort list

- reverse() → Reverse list

- copy() → Shallow copy

Example

num_list = [3, 1, 4, 2] num_list.sort() # [1, 2, 3, 4] num_list.reverse() # [4, 3, 2, 1]

==
==

🔹 TUPLE (Ordered | Immutable | Allows Duplicates)

===

==

✅ Creating a Tuple

my_tuple = (10, 20, 30, 10)

✅ Accessing Elements

second_item = my_tuple[1] # 20

✅ Tuple Methods

- count(x) → Number of occurrences of x

- index(x) → First index of x

✅ Tuple Unpacking

a, b, c, d = my_tuple

===

==

🔹 SET (Unordered | Mutable | No Duplicates)

===

==

✅ Creating a Set

my_set = {1, 2, 3, 4, 2} # Duplicate 2 removed

✅ Adding & Removing

my_set.add(5) # Adds 5 my_set.remove(3) # Removes 3 my_set.discard(10) # Removes 10 if exists popped = my_set.pop() # Removes random item my_set.clear() # Empties set

✅ Set Operations

set1 = {1, 2, 3} set2 = {3, 4, 5} union = set1.union(set2) # {1, 2, 3, 4, 5} intersection = set1.intersection(set2) # {3} difference = set1.difference(set2) # {1, 2} sym_diff = set1.symmetric_difference(set2) # {1, 2, 4, 5}

✅ Set Methods

- add(x) → Add item

- remove(x) → Remove item (error if not present)

- discard(x) → Remove if exists

- pop() → Remove random item

- clear() → Empty set

- union() → Merge sets

- intersection() → Common elements

- difference() → Unique to first set

- symmetric_difference() → Uncommon in both

- issubset() / issuperset() / isdisjoint()

===

🔹 DICTIONARY (Key-Value | Mutable | Keys Unique)

===

==

✅ Creating a Dictionary

my_dict = {"name": "Alice", "age": 25, "city": "London"}

✅ Accessing Values

name = my_dict["name"]

✅ Safe Access

name_safe = my_dict.get("name") unknown = my_dict.get("country", "Not Found")

✅ Updating / Adding

my_dict["age"] = 26 my_dict["country"] = "UK"

✅ Removing Items

del my_dict["city"] popped_value = my_dict.pop("age") last_item = my_dict.popitem() # Removes last inserted item

✅ Dictionary Methods

- get(key[, default]) → Safe access

- keys() → All keys

- values() → All values

- items() → Key-value pairs

- update(dict2) → Merge another dictionary

- pop(key) → Remove key

- popitem() → Remove last inserted item

- clear() → Empty dictionary

- copy() → Shallow copy

✅ Nested Dictionary

student = { "name": "John", "age": 22, "courses": ["Math", "Science"], "grades": {"Math": 90, "Science": 85} }

math_score = student["grades"]["Math"] # 90

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

+ Code + Text