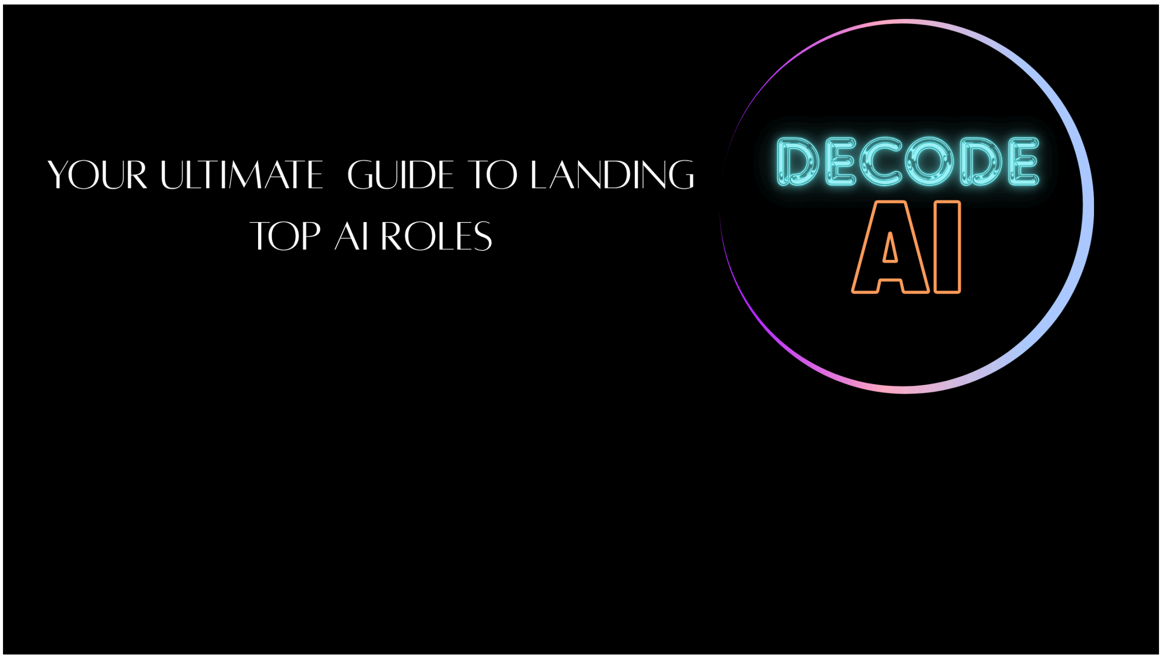

In this session, we are going to learn the following Containers data types:

1. ✅ Tuple
2. ✅ Set
3. ✅ Dictionary

1. Tuples in Python

In Python, **tuples** are similar to lists but they are **immutable** — meaning they **cannot be changed** after creation. Tuples are often used to represent fixed collections of items, such as the days of the week or calendar dates.

What You'll Learn

In this section, we'll explore:

1. Constructing Tuples
2. Basic Tuple Methods
3. Immutability
4. When to Use Tuples ?

You'll develop an intuition for using tuples based on your understanding of lists.
While tuples and lists are structurally similar, the key difference is that **tuples are immutable**.

🛠️ Constructing Tuples

Tuples are created using **parentheses** `()` with elements separated by **commas**.

📌 Example:

```python
# Creating a tuple
my_tuple = (1, 2, 3)

# Tuple with different data types
mixed_tuple = (10, "Python", True)
```

Hands-on Time

In [2]:

```
#Homogenous types
tuple_of_int = (1,2,3)
tuple_of_float = (100.1,200.1,300.0)
tuple_of_string = ("India", "Is", "Great")

print(f'Tuple of Int - {tuple_of_int}')
print(f'Tuple of Float - {tuple_of_float}')
print(f'Tuple of String - {tuple_of_string}')
```

```
Tuple of Int - (1, 2, 3)
Tuple of Float - (100.1, 200.1, 300.0)
Tuple of String - ('India', 'Is', 'Great')
```

In [3]:

```
#Heterogenous types
heterogenous_tuple_type = (1, 100.1, "India", 300.0)

print(f'Heterogenous Tuple Type - {heterogenous_tuple_type}')
```

```
Heterogenous Tuple Type - (1, 100.1, 'India', 300.0)
```

Indexing and slicing

In [5]:

```
# constructing a tuple
tup = (1, 100.1, "India", 300.0)
```

In [6]:

```
# indexing - access 0th item
tup[0]
```

Out[6]:

```
1
```

In [7]:

```
# indexing - access last item
print(tup[3])
print(tup[-1])
```

```
300.0
300.0
```

In [9]:

```
# slicing - access all item starting from index 1
tup[1:]
```

Out[9]:

```
(100.1, 'India', 300.0)
```

In [11]:

```
# slicing - access alternate items in list like 0th,2nd,4th etc
tup[::2]
```

Out[11]:

```
(1, 'India')
```

In [10]:

```
# slicing - tuple items in reverse
tup[::-1]
```

Out[10]:

```
(300.0, 'India', 100.1, 1)
```

Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Let's see two samples of tuple built-in methods:

In [14]:

```
# Use .index to enter a value and return the index
tup.index("India")
```

Out[14]:

```
2
```

In [16]:

```
# Use .count to count the number of times a value appears
tup.count('India')
```

Out[16]:

```
1
```

In [17]:

```
# try to modify tuple values
tup[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 2
      1 # try to modify tuple values
----> 2 tup[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

In [18]:

```
# try to modify tuple values
t.append('nope')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[18], line 2
      1 # try to modify tuple values
----> 2 t.append('nope')

NameError: name 't' is not defined
```

2. Sets in Python

In Python, a **set** is an **unordered collection of unique elements**. Sets are mutable, but the elements they contain must be **immutable (hashable)**.

Sets are useful when you want to **store multiple items**, but only care about **unique values**, such as removing duplicates or performing set operations (union, intersection, etc.).

Key Features of Sets

- **Unordered**: Elements have no defined order.
- **No Duplicate Items**: Automatically removes duplicates.
- **Mutable**: You can add or remove items.
- **Iterable**: Can loop through sets.
- **Unindexed**: Elements cannot be accessed using an index.

Creating a Set

You can create a set using the `set()` constructor or with curly braces `{}`.

Example:

```
# Using curly braces
my_set = {1, 2, 3, 4}

# Using the set() function
another_set = set([1, 2, 2, 3, 4])  # Duplicates are removed

print(my_set)        # Output: {1, 2, 3, 4}
print(another_set)   # Output: {1, 2, 3, 4}
```

NOTE : Sets cannot contain mutable (unhashable) elements like lists or other sets

Example

```
invalid_set = {1, [2, 3]}  # ❌ TypeError: unhashable type: 'list'
invalid_set = {1, {2, 3}}  # ❌ TypeError: unhashable type: 'dict'
```

Hands-On Time

In [23]:

```
#construct an empty set
x = set()
```

In [24]:

```
# We add to sets with the add() method
x.add(1)
```

In [25]:
```
#Show
x
```

Out[25]:
```
{1}
```

In [26]:
```
# Add a different element
x.add(2)
```

In [27]:
```
#show
x
```

Out[27]:
```
{1, 2}
```

In [28]:
```
# Try to add the same element
x.add(1)
```

In [30]:
```
#Show
x
```

Out[30]:
```
{1, 2}
```

it won't place another 1 there as a set is only concerned with unique elements

In [31]:
```
# Create a list with repeats
l = [1,1,2,2,3,4,5,6,1,1]
```

In [32]:
```
# Cast as set to get unique values
set(l)
```

Out[32]:
```
{1, 2, 3, 4, 5, 6}
```

In [33]:
```
# update(iterable) - Adds multiple elements from another iterable (list, set, tuple, etc.).
s = {1, 2}
s.update([3, 4])  # s becomes {1, 2, 3, 4}
s
```

Out[33]:
```
{1, 2, 3, 4}
```

In [34]:
```
# remove(elem) - Removes the specified element. ❌ Raises KeyError if the element is not found.
s = {1, 2, 3}
s.remove(2)      # s becomes {1, 3}
s
```

Out[34]:
```
{1, 3}
```

In [35]:
```
# discard(elem) - Removes the specified element if it exists. ✅ No error if not found.
s = {1, 2, 3}
s.discard(4)     # No error; s remains {1, 2, 3}
s
```

Out[35]:
```
{1, 2, 3}
```

In [36]:
```
# pop() - Removes and returns a random element from the set.

s = {10, 20, 30}
s.pop()          # Randomly removes one element
s
```

Out[36]:
```
{20, 30}
```

In [37]:
```
# clear() - Removes all elements from the set.

s = {1, 2, 3}
s.clear()        # s becomes set()
s
```

Out[37]:
```
set()
```

set operations

In [38]:

```
# union(other_set) or | => Returns a new set with elements from both sets.
a = {1, 2}
b = {2, 3}
print(a.union(b))     # {1, 2, 3}
print(a | b)          # {1, 2, 3}
```

```
{1, 2, 3}
{1, 2, 3}
```

In [39]:

```
# intersection(other_set) or & => Returns common elements.
a = {1, 2, 3}
b = {2, 3, 4}
print(a.intersection(b))  # {2, 3}
print(a & b)              # {2, 3}
```

```
{2, 3}
{2, 3}
```

In [40]:

```
# difference(other_set) or - => Returns elements in the first set but not in the second.
a = {1, 2, 3}
b = {2, 3}
print(a.difference(b))    # {1}
print(a - b)              # {1}
```

```
{1}
{1}
```

In [41]:

```
# symmetric_difference(other_set) or ^ => Returns elements that are in either of the sets but not both
a = {1, 2, 3}
b = {3, 4}
print(a.symmetric_difference(b))  # {1, 2, 4}
print(a ^ b)                      # {1, 2, 4}
```

```
{1, 2, 4}
{1, 2, 4}
```

In [42]:

```
# issubset(other_set) => Checks if all elements of this set are in the other set.
a = {1, 2}
b = {1, 2, 3}
print(a.issubset(b))   # True
```

```
True
```

In [43]:

```
# issuperset(other_set) => Checks if the set contains all elements of the other set.
a = {1, 2, 3}
b = {2, 3}
print(a.issuperset(b)) # True
```

```
True
```

In [44]:

```
# isdisjoint(other_set) => Checks if two sets have no common elements.
a = {1, 2}
b = {3, 4}
print(a.isdisjoint(b)) # True
```

```
True
```

3. Dictionary in Python

In Python, a **dictionary** is a powerful built-in data structure that allows you to store and manage **data in key-value pairs**.

What is a Dictionary?

A **dictionary** is a collection that is:

- **Unordered** (prior to Python 3.7), **insertion-ordered** (from Python 3.7+)
- **Mutable**: You can change, add, or remove items
- **Indexed by keys**, not by numerical position
- Made up of **unique keys** and their **associated values**

Syntax

Dictionaries are defined using **curly braces** `{}` or the `dict()` constructor.

```
# Using curly braces
person = {
    "name": "Alice",
    "age": 25,
    "city": "Delhi"
}

# Using dict() constructor
employee = dict(name="Bob", department="IT", salary=75000)
```

Properties of Dictionary

Property	Supported	Notes
Key-Value Storage	✅	Fast and flexible
Mutable	✅	Can modify after creation
Insertion Ordered	✅	From Python 3.7+
Duplicate Keys	❌	Last occurrence is stored
Indexed by Position	❌	Access by key only
Nested Structures	✅	Values can be lists, dicts, etc.
Iterable	✅	Can iterate over keys, values, or items
Hash-Based Lookup	✅	Fast retrieval using keys (O(1) average case)

In [46]:

```
d = {}
d["ds"] = 1
d["bd"] = 2
d["c1"] = 3

print(d)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

```
{'ds': 1, 'bd': 2, 'c1': 3}
```

Hands-On Time

In [1]:

```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

In [2]:

```
# Call values by their key
my_dict['key2']
```

Out[2]:

```
'value2'
```

In [15]:

```
# homogenous data examples 1
marks = {"Ramesh":60, "Mahesh":90, "Bhupesh":80}
# show
marks
```

Out[15]:

```
{'Ramesh': 60, 'Mahesh': 90, 'Bhupesh': 80}
```

In [16]:

```
# homogenous data examples 2
grade = {"Ramesh":"B", "Mahesh":"A", "Bhupesh":"B"}
# show
grade
```

Out[16]:

```
{'Ramesh': 'B', 'Mahesh': 'A', 'Bhupesh': 'B'}
```

In [17]:

```
# heterogenous data examples
person = {
    "name": "Sanjeev",      # str
    "age": 28,              # int
    "is_student": False,    # bool
    "skills": ["Python", "ML"]  # list
}
```

```
#show
person
```

Out[17]:

```
{'name': 'Sanjeev', 'age': 28, 'is_student': False, 'skills': ['Python', 'ML']}
```

In [19]:

```
# Creating a dictionary - Approach 1
marks = {"Ramesh":100}

#show
marks
```

Out[19]:

```
{'Ramesh': 100}
```

In [18]:

```
# Creating a dictionary - Approach 2
marks = {}
marks["Ramesh"] = 100

#show
marks
```

Out[18]:

```
{'Ramesh': 100}
```

In [20]:

```
# Creating a dictionary - Approach 2
marks = dict()
marks["Ramesh"] = 100

#show
marks
```

Out[20]:

```
{'Ramesh': 100}
```

In [21]:

```
# Properties of dictionaries
marks = {"Ramesh":60, "Mahesh":90, "Bhupesh":80}
```

In [22]:

```
# keys
marks.keys()
```

Out[22]:

```
dict_keys(['Ramesh', 'Mahesh', 'Bhupesh'])
```

In [23]:

```
#keys properties - dict_keys : dict_keys is iterable, just like a list
print(type(marks.keys()))
```

```
<class 'dict_keys'>
```

In [24]:

```
#iterate over dict keys
for key in marks.keys():
    print(key)
```

```
Ramesh
Mahesh
Bhupesh
```

In [27]:

```
# can convert dict_keys class to other types
print(list(marks.keys()))
print(set(marks.keys()))
```

```
['Ramesh', 'Mahesh', 'Bhupesh']
{'Bhupesh', 'Mahesh', 'Ramesh'}
```

In [28]:

```
#keys properties - dict_keys : dict_values is iterable, just like a list
print(type(marks.values()))
```

```
<class 'dict_values'>
```

In [29]:

```
#iterate over dict values
for value in marks.values():
    print(value)
```

```
60
90
80
```

In [30]:

```
# can convert dict_values class to other types
print(list(marks.values()))
print(set(marks.values()))
```

```
[60, 90, 80]
{80, 90, 60}
```

In [32]:
```python
# not supported keys - non hashable keys - part 1
my_dict = {
    (1,2,3): 23,
    "Name": "Ramesh"
}
my_dict
```

Out[32]:
```
{(1, 2, 3): 23, 'Name': 'Ramesh'}
```

In [33]:
```python
# not supported keys - non hashable keys - part 2
my_dict = {
    [1,2,3]: 23,
    "Name": "Ramesh"
}
my_dict
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 2
      1 # not supported keys - non hashable keys - part 2
----> 2 my_dict = {
      3     [1,2,3]: 23,
      4     "Name": "Ramesh"
      5 }
      6 my_dict

TypeError: unhashable type: 'list'
```

In [34]:
```python
# not supported keys - non hashable keys - part 3
my_dict = {
    {1,2,3}: 23,
    "Name": "Ramesh"
}
my_dict
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 2
      1 # not supported keys - non hashable keys - part 3
----> 2 my_dict = {
      3     {1,2,3}: 23,
      4     "Name": "Ramesh"
      5 }
      6 my_dict

TypeError: unhashable type: 'set'
```

indexing examples

In [35]:
```python
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [36]:
```python
#Let's call items from the dictionary
my_dict['key3']
```

Out[36]:
```
['item0', 'item1', 'item2']
```

In [55]:
```python
# accessing a non-existent element
my_dict['key4']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[55], line 2
      1 # accessing a non-existent element
----> 2 my_dict['key4']

KeyError: 'key4'
```

In [56]:
```python
# Accessing with Safety : dict.get(key[, default])
print(my_dict.get("key4"))       # 1
print(my_dict.get("key4", 0))    # 0
```

```
None
0
```

In [37]:

```python
# Can call an index on that value
my_dict['key3'][0]
```

Out[37]:

```
'item0'
```

In [38]:

```python
#Can then even call methods on that value
my_dict['key3'][0].upper()
```

Out[38]:

```
'ITEM0'
```

Removing Elements

In [58]:

```python
marks = {"Ramesh":60, "Mahesh":90, "Bhupesh":80}
```

In [62]:

```python
# dict.pop(key[, default]) => Removes and returns the value of the given key. Raises error if key is
marks.pop("Sanjeev")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[62], line 2
      1 # dict.pop(key[, default]) => Removes and returns the value of the given key. Raises error if
key is not found unless a default is provided.
----> 2 marks.pop("Sanjeev")

KeyError: 'Sanjeev'
```

In [63]:

```python
marks.pop("Sanjeev",-1)
```

Out[63]:

```
-1
```

In [64]:

```python
# dict.popitem() => Removes and returns the last inserted key-value pair.
marks.popitem()
```

Out[64]:

```
('Bhupesh', 80)
```

In [65]:

```python
marks
```

Out[65]:

```
{'Ramesh': 60, 'Mahesh': 90}
```

In [67]:

```python
# dict.clear() => Removes all key-value pairs from the dictionary.
marks.clear()
```

In [68]:

```python
#marks
marks
```

Out[68]:

```
{}
```

In [69]:

```python
# dict.update(other_dict)
my_dict = {"a": 1}
my_dict.update({"b": 2, "c": 3})
```

In [70]:

```python
my_dict
```

Out[70]:

```
{'a': 1, 'b': 2, 'c': 3}
```

modifying values of a key as well

In [41]:

```python
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [42]:

```python
my_dict['key1']
```

Out[42]:

```
123
```

In [43]:

```python
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

In [44]:

```python
#Check
my_dict['key1']
```

```
Out[44]: 0
```

```
In [45]: # Set the object equal to itself plus 123
         my_dict['key1'] += 123
         my_dict['key1']
```

```
Out[45]: 123
```

```
In [46]: # creating a new key in a dictionary
```

```
In [47]: d = {}
```

```
In [48]: # Create a new key through assignment
         d['animal'] = 'Dog'
```

```
In [49]: # Can do this with any object
         d['answer'] = 42
```

```
In [50]: #Show
         d
```

```
Out[50]: {'animal': 'Dog', 'answer': 42}
```

```
In [ ]: # dictionaries nesting
```

```
In [51]: d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

```
In [52]: # Keep calling the keys
         d['key1']['nestkey']['subnestkey']
```

```
Out[52]: 'value'
```

Checking Membership

```
In [71]: marks = {"Ramesh":60, "Mahesh":90, "Bhupesh":80}
```

```
In [73]: # check for a key - m1
         "Ramesh" in marks
```

```
Out[73]: True
```

```
In [74]: # check for a key - m2
         "Ramesh" in marks.keys()
```

```
Out[74]: True
```

```
In [77]: # check for a value
         90 in marks.values()
```

```
Out[77]: True
```

Dictionary Comprehension

Just like List Comprehensions, Dictionary Data Types also support their own version of comprehension for quick creation. It is not as commonly used as List Comprehensions, but the syntax is:

```
In [ ]: # Problem Statement: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

```
In [53]: # Trivial Approach
         my_dict = {}
         for i in range(10):
             my_dict[i] = i**2

         my_dict
```

```
Out[53]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

```
In [54]: # Smart approach - Dictionary comprehension
         my_dict = {i:i**2 for i in range(10)}
         my_dict
```

```
Out[54]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

🙏 Thank you for learning with us!

— Team 🚀 Decode-AI