Tuples in python

Tuples are ordered collection of data items. They store multiple items in a single variable. Tuple items are separated by commas and enclosed within round brackets (). Tuples are unchangeable meaning we can not alter them after creation.

Example:

```
tuple1 = (1,2,2,3,5,4,6)
tuple2 = ("Red", "Green", "Blue")
print(tuple1)
print(tuple2)
```

Output:

(1, 2, 2, 3, 5, 4, 6)
('Red', 'Green', 'Blue')\

Tuple Indexes

Each item/element in a tuple has its own unique index. This index can be used to access any particular item from the tuple. The first item has index [0], second item has index [1], third item has index [2] and so on.

Example:

```
country = ("Spain", "Italy", "India",)
#            [0]      [1]      [2]
```

❖ **Accessing tuple items:**

I. Positive Indexing:

As we have seen that tuple items have index, as such we can access items using these indexes.

Example:

```
country = ("Spain", "Italy", "India",)
#            [0]      [1]      [2]
print(country[0])
print(country[1])
```

Output:

Spain
Italy

II. Negative Indexing:

Example:

```
country = ("Spain", "Italy", "India", "England", "Germany")
#            [0]      [1]      [2]       [3]         [4]
print(country[-1]) # Similar to print(country[len(country) -
1])
```

TECHVISION-TECHNOLOGY

```
print(country[-3])
```
Output:
Germany
India

III. Check for item:
We can check if a given item is present in the tuple. This is done using the in keyword.

Example 1:
```
country = ("Spain", "Italy", "India", "England", "Germany")
if "Germany" in country:
    print("Germany is present.")
else:
    print("Germany is absent.")
```
Output:
Germany is present.

IV. Range of Index:
You can print a range of tuple items by specifying where do you want to start, where do you want to end and if you want to skip elements in between the range.

Syntax:
```
Tuple[start : end : jumpIndex]
```
Note: jump Index is optional. We will see this in given examples.

Example 1: Printing elements within a particular range:
```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
print(animals[3:7])     #using positive indexes
print(animals[-7:-2])   #using negative indexes
```
Output:
('mouse', 'pig', 'horse', 'donkey')
('bat', 'mouse', 'pig', 'horse', 'donkey')

Example 2: Printing all element from a given index till the end
```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
print(animals[4:])      #using positive indexes
print(animals[-4:])     #using negative indexes
```

Output:
('pig', 'horse', 'donkey', 'goat', 'cow')
('horse', 'donkey', 'goat', 'cow')
When no end index is provided, the interpreter prints all the values till the end.

Example 3: printing all elements from start to a given index
```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
```

TECHVISION TECHNOLOGY

```
print(animals[:6])      #using positive indexes
print(animals[:-3])     #using negative indexes
```

Output:

('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example 4: Print alternate values

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
print(animals[::2])     #using positive indexes
print(animals[-8:-1:2]) #using negative indexes
```

Output:

('cat', 'bat', 'pig', 'donkey', 'cow')
('dog', 'mouse', 'horse', 'goat')

❖ Manipulating Tuples

Tuples are immutable, hence if you want to add, remove or change tuple items, then first you must convert the tuple to a list. Then perform operation on that list and convert it back to tuple.

Example:

```
countries = ("Spain", "Italy", "India", "England", "Germany")
temp = list(countries)
temp.append("Russia")      #add item
temp.pop(3)                #remove item
temp[2] = "Finland"        #change item
countries = tuple(temp)
print(countries)
```

Output:

('Spain', 'Italy', 'Finland', 'Germany', 'Russia')

❖ Tuple methods

As tuple is immutable type of collection of elements it have limited built in methods.They are explained below

➢ count() Method

The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax:

```
tuple.count(element)
```

Example

```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.count(3)
print('Count of 3 in Tuple1 is:', res)
```

Output

TECHVISION TECHNOLOGY

➢ index() method

The Index() method returns the first occurrence of the given element from the tuple.

Syntax:

```
tuple.index(element, start, end)
```

Note: This method raises a ValueError if the element is not found in the tuple.

Example:

```
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple.index(3)
print('First occurrence of 3 is', res)
```

Output

3

String Formatting in Python

String formatting is a way to **insert variables or values** into strings.

1. Using format() method (Python 3 and above)

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name,
age))
```

With index numbers:

```
print("My name is {0} and I am {1} years old. {0} is learning
Python.".format(name, age))
```

With named placeholders:

```
print("My name is {n} and I am {a} years
old.".format(n="Alice", a=25))
```

2. Using f-strings (Python 3.6+)

```
name = "Bob"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

You can use expressions inside f-strings:

```
print(f"Next year I will be {age + 1} years old.")
```

3. Using % operator (Old method, still supported)

TECHVISION-TECHNOLOGY

```
name = "Charlie"
age = 28
print("My name is %s and I am %d years old." % (name, age))
```

What is a Set in Python?

A **set** is an unordered, unindexed collection of **unique elements**.
It is **mutable** (can be changed) but cannot contain **mutable items** (like lists or dictionaries).
Defined using curly braces `{}` or the `set()` function.

```
my_set = {1, 2, 3, 4}
another_set = set([3, 4, 5])
```

Key Features of Sets:

Feature	Description
Unordered	No indexing, elements have no order
No Duplicates	All elements are unique
Mutable	You can add or remove items

Creating a Set:

```
# Empty set (use set(), not {})
empty_set = set()
# Set with values
fruits = {'apple', 'banana', 'mango'}
```

Commonly Used Set Methods:

Method	Description	Example
`add()`	Adds an element to the set	`my_set.add(10)`
`remove()`	Removes a specific element (error if not found)	`my_set.remove(2)`
`discard()`	Removes element (no error if not found)	`my_set.discard(5)`
`pop()`	Removes and returns a random item	`item = my_set.pop()`
`clear()`	Empties the entire set	`my_set.clear()`
`copy()`	Returns a shallow copy	`new_set = my_set.copy()`

TECHVISION-TECHNOLOGY

Method	Description	Example
`update()`	Adds multiple elements from another set/list	`my_set.update([7, 8])`

Set Operations:

Operation	Symbol / Method	Description
Union	`` `	`orset1.union(set2)` `
Intersection	`& or set1.intersection(set2)`	Common elements in both sets
Difference	`- or set1.difference(set2)`	Elements in `set1` but not in `set2`
Symmetric Difference	`^ or set1.symmetric_difference(set2)`	Elements in either set, but not both

Examples:

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

print(a | b)    # Union: {1, 2, 3, 4}
print(a & b)    # Intersection: {2, 3}
print(a - b)    # Difference: {1}
print(a ^ b)    # Symmetric Difference: {1, 4}
```

Set Membership

```
if 2 in my_set:
    print("2 is present")

if 9 not in my_set:
    print("9 is not present")
```

TECHVISION-TECHNOLOGY

What is a Dictionary in Python?

A **dictionary** is an **unordered collection** of **key-value pairs**.
Each key is **unique** and maps to a value.
Defined using curly braces `{}` with keys and values separated by a colon `:`

```
student = {'name': 'John', 'age': 21, 'course': 'Python'}
```

Key Features of Dictionaries:

Feature	Description
Key-Value	Each element is a key-value pair
Unordered	Order is preserved from Python 3.7+

Feature	Description
Mutable	You can change, add, or remove items
No Duplicates	Keys must be unique

Creating a Dictionary:

```
# Basic dictionary
person = {'name': 'Alice', 'age': 25}

# Empty dictionary
empty_dict = {}

# Using dict() function
data = dict(name='Bob', age=30)
```

Accessing and Modifying Items:

```
# Access value by key
print(person['name'])          # Output: Alice

# Add or update a value
person['age'] = 26             # Updates age
person['city'] = 'London'      # Adds new key-value pair

# Using get() (returns None if key doesn't exist)
print(person.get('gender'))    # Output: None
```

Removing Items:

Method	Description	Example
`pop(key)`	Removes item by key	`person.pop('age')`
`popitem()`	Removes the last inserted item	`person.popitem()`
`del`	Deletes key or entire dictionary	`del person['name']`
`clear()`	Removes all items	`person.clear()`

Looping Through a Dictionary:

```
# Loop through keys
for key in person:
    print(key)

# Loop through values
for value in person.values():
    print(value)

# Loop through key-value pairs
for key, value in person.items():
    print(key, value)
```

TECHNISON-TECHNOLOGY

Useful Dictionary Methods:

Method	Description
`keys()`	Returns all keys
`values()`	Returns all values
`items()`	Returns all key-value pairs as tuples
`get(key)`	Returns value for key, or None if not found
`update(dict2)`	Updates dictionary with another dictionary
`copy()`	Returns a shallow copy
`fromkeys(keys, val)`	Creates new dict with keys and a default value
`setdefault()`	Returns value of key, sets default if not found

Example: Full Dictionary Workflow:

```
student = {
    'name': 'Rahul',
    'roll': 101,
    'subject': 'Math'
}

# Add new key
student['marks'] = 95

# Update existing value
student['subject'] = 'Science'

# Access value
print(student.get('name'))

# Loop through keys and values
for k, v in student.items():
    print(f"{k}: {v}")
```

TECHVISION-TECHNOLOGY

How to Raise a Custom Error

You can use Python's built-in exceptions with a custom message:

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative")
```

Short-hand `if-else` in Python

Python allows a concise way to write `if-else` in one line, called a **ternary conditional operator**.

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)

# Output: Adult
```

`if __name__ == "__main__"` in Python — Explained Simply:

```
if __name__ == "__main__":
    # run this code
```

This line checks whether the script is being **run directly** or **imported as a module**.

Why is it used?

To **prevent certain code from running when the file is imported** elsewhere.

How it works:

When you run a Python file directly, Python sets the special variable `__name__` to `"__main__"`.

When you **import the file into another Python script**, `__name__` becomes the **name of the module** (i.e., the file name, not `"__main__"`).

Example:

```
# file: myscript.py

def greet():
    print("Hello!")

if __name__ == "__main__":
    greet()  # This will only run if you run myscript.py
directly

python myscript.py
```

Output:

```
Hello!
```

But if you do:

TECHVISION-TECHNOLOGY

```
import myscript
```

Nothing will print automatically.

Why use it?

- Helps in writing **reusable** and **testable** code.
- Keeps test/demo code separate from functions/classes.

`Enumerate` Function in Python

The `enumerate()` function adds a **counter** (index) to an **iterable** (like a list, tuple, or string) and returns it as an `enumerate` object.

Syntax:

```
enumerate(iterable, start=0)
```

- `iterable`: Any iterable object (like list, tuple, string, etc.)
- `start`: (Optional) The index to start counting from. Default is 0.

Example 1: With List:

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

Example 2: Starting from 1:

```
for index, fruit in enumerate(fruits, start=1):
    print(index, fruit)
```

Output:

```
1 apple
2 banana
3 cherry
```

TECHVISION-TECHNOLOGY

How importing in python works

Importing in Python is the process of loading code from a Python module into the current script. This allows you to use the functions and variables defined in the module in your current script, as well as any additional modules that the imported module may depend on.

Once a module is imported, you can use any of the functions and variables defined in the module by using the dot notation. For example, to use the sqrt function from the math module, you would write:

```
import math
result = math.sqrt(9)
print(result)  # Output: 3.0
```

from keyword:

You can also import specific functions or variables from a module using the from keyword. For example, to import only the sqrt function from the math module, you would write:

```
from math import sqrt

result = sqrt(9)
print(result)  # Output: 3.0
```

You can also import multiple functions or variables at once by separating them with a comma:

```
from math import sqrt, pi

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

importing everything:

It's also possible to import all functions and variables from a module using the * wildcard. However, this is generally not recommended as it can lead to confusion and make it harder to understand where specific functions and variables are coming from.

```
from math import *

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

Python also allows you to rename imported modules using the as keyword. This can be useful if you want to use a shorter or more descriptive name for a module, or if you want to avoid naming conflicts with other modules or variables in your code.

TECHVISION-TECHNOLOGY

The "as" keyword:

```
import math as m

result = m.sqrt(9)
print(result)  # Output: 3.0

print(m.pi)  # Output: 3.141592653589793
```

The dir function:

Finally, Python has a built-in function called dir that you can use to view the names of all the functions and variables defined in a module. This can be helpful for exploring and understanding the contents of a new module.

```
import math

print(dir(math))
```

This will output a list of all the names defined in the math module, including functions like sqrt and pi, as well as other variables and constants.

Python OS Module

The `os` module in Python provides a way to interact with the **operating system**.
It allows you to perform tasks like creating files/folders, navigating directories, and managing environment variables.

1. os.getcwd() => Get Current Working Directory

```
import os
print(os.getcwd())
```

Explanation:
This function shows where your Python script is running from.

2. os.mkdir() => Create a New Directory

```
os.mkdir("new_folder")
```

Explanation:
Creates a folder named new_folder in the current directory.
If it already exists, it will give an error.

3. os.makedirs() => Create Directory with Subfolders

```
os.makedirs("parent/child/grandchild")
```

Explanation:
Creates all folders in the path if they don't exist.
This will create parent, then child inside it, and grandchild inside that.

TECHVISION-TECHNOLOGY

4. os.chdir() => Change Directory

```
os.chdir("C:/Users/Username/Documents")
print(os.getcwd())
```

Explanation:
This changes the current working directory to the one you provide.
After that, os.getcwd() will show this new location.

5. os.listdir() => List Files and Folders in Directory

```
files = os.listdir()
print(files)
```

Explanation:
Lists everything (files and folders) in the current directory.
You can also pass a path like os.listdir("C:/Users").

6. os.rename() => Rename File or Folder

```
os.rename("old_name.txt", "new_name.txt")
```

Explanation:
Changes the name of a file or folder.

7. os.remove() => Delete a File

```
os.remove("unwanted.txt")
```

Explanation:
Deletes the file named unwanted.txt.
Make sure it's a file, not a folder.

8. os.rmdir() => Remove an Empty Folder

```
os.rmdir("empty_folder")
```

Explanation:
Removes a folder, but it must be **empty**.
Use os.removedirs() for recursive removal.

9. os.path.exists() => Check if Path Exists

```
if os.path.exists("data.txt"):
    print("File exists")
else:
    print("Not found")
```

Explanation:
Checks whether the file or folder is present or not.

10. os.path.isfile() & os.path.isdir()

```
print(os.path.isfile("file.txt"))   # True if it is a file
print(os.path.isdir("foldername"))  # True if it is a folder
```

TECHVISION-TECHNOLOGY

11. os.environ and os.getenv()

```
print(os.environ)  # Shows all environment variables

# Get specific one
print(os.getenv('PATH'))
```

Explanation:
Used to read system environment variables like PATH, HOME, etc.

Local Scope vs Global Scope in Python

1. Global Scope

- **Definition**: A variable with global scope is defined outside all functions and is accessible throughout the program.
- **Lifespan**: It exists as long as the program runs.
- **Access**: Can be accessed from inside and outside functions.

Example:

```
x = 10  # Global variable

def show():
    print(x)  # Accessing global variable

show()
print(x)  # Also accessible here
```

2. Local Scope

- **Definition**: A variable with local scope is defined inside a function and can only be used inside that function.
- **Lifespan**: It exists only while the function is executing.
- **Access**: Not accessible outside the function.

Example:

```
def show():
    y = 5  # Local variable
    print(y)

show()
# print(y)  #  This will cause an error (y is not defined here)
```

Using Global Keyword

If you want to **modify** global variable inside a function, you must use the `global` keyword.

TECHVISION-TECHNOLOGY

Example:

```
x = 10

def update():
    global x
    x = 20  # Modifies the global variable
    print("Inside:", x)

update()
print("Outside:", x)
```

Python File Handling Modes

Basic Syntax to Open a File

```
file = open("filename.txt", "mode"
```

File Modes in Python

Mode	Name	Description
'r'	**Read**	Opens the file for reading. File **must exist**.
'w'	**Write**	Opens the file for writing. **Overwrites** the file if it exists, or creates a new file.
'a'	**Append**	Opens the file for appending. Adds content to the end without deleting existing data.
'x'	**Exclusive Creation**	Creates a new file. Fails if the file already exists.
't'	**Text Mode**	Default mode. File is handled as text.
'b'	**Binary Mode**	File is handled as binary (images, videos, etc.).

Combined Modes

Mode	Description
'rt'	Read text (default)
'rb'	Read binary
'wt'	Write text (overwrites)
'wb'	Write binary (overwrites)
'at'	Append text
'ab'	Append binary
'r+'	Read and write (file must exist)
'w+'	Write and read (file is overwritten)
'a+'	Append and read

TECHNOTECHNOLOGY

Examples

1. Reading a File

```
f = open("data.txt", "r")
content= f.read())
print(content)
f.close()
```

2. Writing to a File

```
f = open("data.txt", "w")
f.write("Hello, world!")
f.close()
```

3. Appending to a File

```
f = open("data.txt", "a")
f.write("\nNew line added.")
f.close()
```

4. Read + Write

```
f = open("data.txt", "r+")
data = f.read()
print(data)
f.write("\nExtra line.")
f.close()
```

Tips

Always use `close()` to close files or use:

```
with open("file.txt", "r") as f:
    data = f.read()
```

with automatically closes the file.

TECHVISION-TECHNOLOGY

Some more methods in file handling

readlines() method:

The readline() method reads a single line from the file. If we want to read multiple lines, we can use a loop.

```
f = open('myfile.txt', 'r')
while True:
    line = f.readline()
    if not line:
        break
```

```
    print(line)
```

The readlines() method reads all the lines of the file and returns them as a list of strings.

writelines() method:

The writelines() method in Python writes a sequence of strings to a file. The sequence can be any iterable object, such as a list or a tuple.
Here's an example of how to use the writelines() method:

```
f = open('myfile.txt', 'w')
lines = ['line 1\n', 'line 2\n', 'line 3\n']
f.writelines(lines)
f.close()
```

This will write the strings in the lines list to the file myfile.txt. The \n characters are used to add newline characters to the end of each string.
Keep in mind that the writelines() method does not add newline characters between the strings in the sequence. If you want to add newlines between the strings, you can use a loop to write each string separately:

```
f = open('myfile.txt', 'w')
lines = ['line 1', 'line 2', 'line 3']
for line in lines:
    f.write(line + '\n')
f.close()
```

seek() function:

The seek() function allows you to move the current position within a file to a specific point. The position is specified in bytes, and you can move either forward or backward from the current position. For example:

```
with open('file.txt', 'r') as f:
  # Move to the 10th byte in the file
  f.seek(10)

  # Read the next 5 bytes
  data = f.read(5)
```

tell() function:

The tell() function returns the current position within the file, in bytes. This can be useful for keeping track of your location within the file or for seeking to a specific position relative to the current position. For example:

```
with open('file.txt', 'r') as f:
  # Read the first 10 bytes
  data = f.read(10)

  # Save the current position
```

TECHVISION-TECHNOLOGY

```
  current_position = f.tell()

  # Seek to the saved position
  f.seek(current_position)
```

truncate() function:

When you open a file in Python using the open function, you can specify the mode in which you want to open the file. If you specify the mode as 'w' or 'a', the file is opened in write mode and you can write to the file. However, if you want to truncate the file to a specific size, you can use the truncate function.
Here is an example of how to use the truncate function:

```
with open('sample.txt', 'w') as f:
  f.write('Hello World!')
  f.truncate(5)

with open('sample.txt', 'r') as f:
  print(f.read())
```

Lambda Function in Python

A **lambda function** is an **anonymous (nameless) function** defined using the `lambda` keyword.
It is used for creating **small, one-line functions** without a name.
Typically used when a short function is needed for a short period of time.

Example 1: Add two numbers

```
add = lambda a, b: a + b
print(add(5, 3))  # Output: 8
```

Example 2: Square of a number

```
square = lambda x: x * x
print(square(4))  # Output: 16
```

Map Filter reduce in python

1. **`map()`** – Apply a function to each item

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4]
squared = list(map(square, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

TECHVISION TECHNOLOGY

2. `filter()` – Keep items that match a condition

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5]
evens = list(filter(is_even, numbers))
print(evens)  # Output: [2, 4]
```

3. `reduce()` – Reduce to a single value

```
from functools import reduce

def multiply(x, y):
    return x * y

numbers = [1, 2, 3, 4]
product = reduce(multiply, numbers)
print(product)  # Output: 24
```

Summary Table (Using Named Functions):

Function	Custom Function Example	What It Does	Output Example
`map()`	`square(x)`	Transforms each item	`[1, 4, 9, 16]`
`filter()`	`is_even(x)`	Filters based on condition	`[2, 4]`
`reduce()`	`multiply(x, y)`	Combines all items into one value	`24`

"is" vs "==" in Python

1. == (Equality Operator)

**Checks if the values of two variables are equal.
It compares data/content stored inside the objects.

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # Output: True
```

Even though `a` and `b` are two different lists (stored in different memory locations), their values are the same, so `==` returns True.

2. is (Identity Operator)

TECHVISION TECHNOLOGY

**Checks if two variables refer to the same object in memory.
It compares the memory address (object identity).

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a is b)  # Output: False
```

`a` and `b` have the same values, but they are different objects in memory. So `is` returns False.

Special Case – Small Integers & Strings

Python caches small integers and strings, so sometimes `is` and `==` both return `True`:

```
x = 5
y = 5
print(x is y)  # Output: True (because of caching)

s1 = "hello"
s2 = "hello"
print(s1 is s2)  # Output: True
```

Summary Table:

Operator	Checks for	Compares	Example Output	Use When
`==`	Equality	Values	`True`	You care about content
`is`	Identity (same object)	Memory addresses	`False`	You care about same object

Final Tip:

Use `==` when comparing values (numbers, lists, strings).
Use `is` when checking if two variables point to the same object (like `None`, singletons).

TECHVISION-TECHNOLOGY