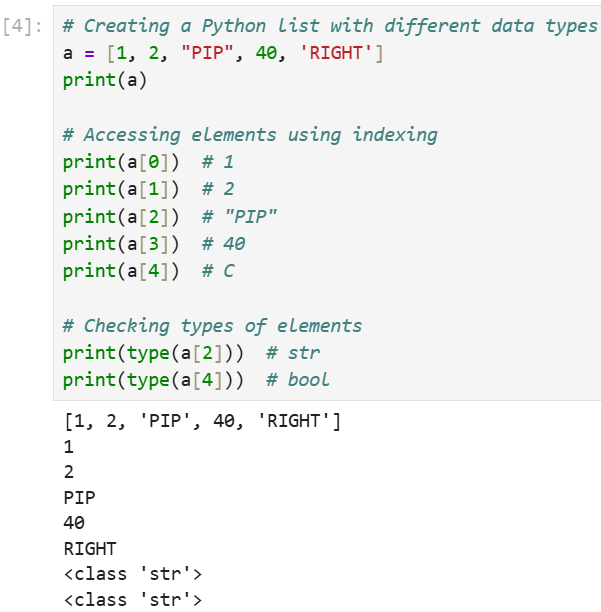
**DATA STRUCTURES & FUNCTIONS**

**PYTHON LISTS**

In Python, a list is a built-in dynamic sized array (automatically grows and shrinks). We can store all types of items (including another list) in a list. A list may contain mixed type of items, this is possible because a list mainly stores references at contiguous locations and actual items maybe stored at different locations.

* List can contain duplicate items.
* List in Python are Mutable. Hence, we can modify, replace or delete the items.
* List are ordered. It maintains the order of elements based on how they are added.
* Accessing items in List can be done directly using their position (index), starting from 0.

**EXAMPLE:**

****

**EXPLANATION:**

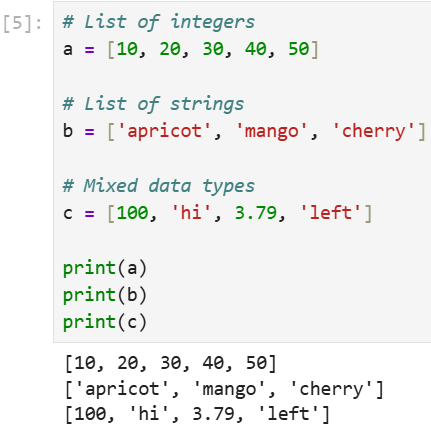
* The list contains a mix of integers (1, 2, 40), a string ("PIP") and a STRING (RIGHT).
* The list is printed and individual elements are accessed using their indexes (starting from 0).
* type(a[2]) confirms "PIP" is a str.
* type(a[4]) confirms “RIGHT” is a str.

***Note:*** Lists Store References, Not Values

Each element in a list is not stored directly inside the list structure. Instead, the list stores references (pointers) to the actual objects in memory. **Example** (from the image representation).

* The list a itself is a container with references (addresses) to the actual values.
* Python internally creates separate objects for 1, 2, "PIP", 40 and RIGHT, then stores their memory addresses inside a.
* This means that modifying an element doesn’t affect other elements but can affect the referenced object if it is mutable

**HOW TO CREATE A LIST:**

****

**LIST LENGTH:**

To determine how many items a list has, use the len() function:

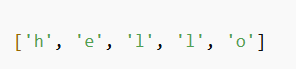


**Using list() Constructor:**

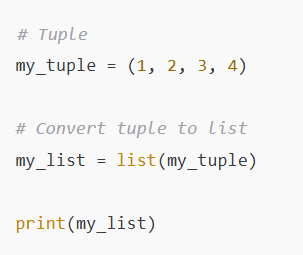
We can also create a list by passing an iterable (like a string, tuple or another list) to **list() function.**

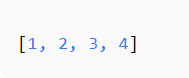
**EXAMPLE 1:**

****

****

**EXAMPLE 2:**

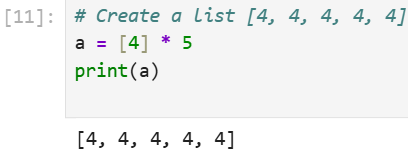
****

****

**Creating List with Repeated Elements**

We can create a list with repeated elements using the multiplication operator.

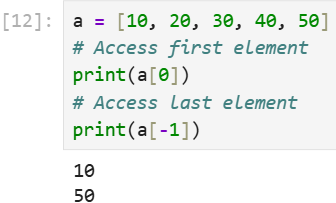
**EXAMPLE:**

****

**Accessing List Elements:**

Elements in a list can be accessed using **indexing**. Python indexes start at **0**, so **a[0]** will access the first element, while**negative indexing** allows us to access elements from the end of the list. Like index -1 represents the last elements of list.

**EXAMPLE:**

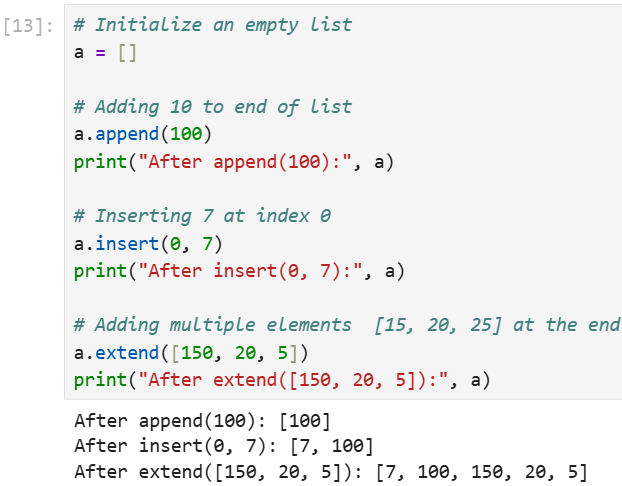
****

**Adding Elements into List:**

We can add elements to a list using the following methods:

* append(): Adds an element at the end of the list.
* extend(): Adds multiple elements to the end of the list.
* insert(): Adds an element at a specific position.

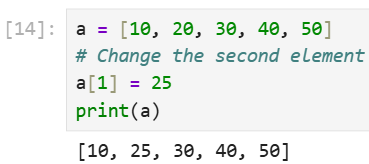
**EXAMPLE:**

****

**Updating Elements into List:**

We can change the value of an element by accessing it using its index.

**EXAMPLE:**

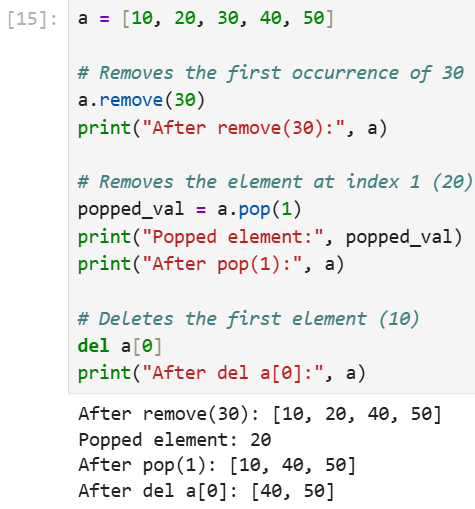
****

**Removing Elements from List:**

We can remove elements from a list using:

* **remove():** Removes the first occurrence of an element.
* **pop():** Removes the element at a specific index or the last element if no index is specified.
* **del statement:** Deletes an element at a specified index.

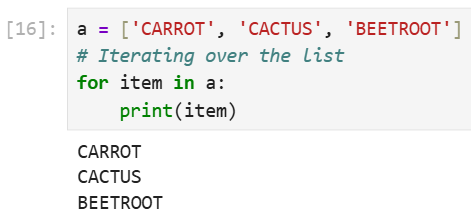
**EXAMPLE:**

****

**Iterating Over Lists:**

We can iterate the Lists easily by using a for loop or other iteration methods. Iterating over lists is useful when we want to do some operation on each item or access specific items based on certain conditions. Let's take an example to iterate over the list using for loop.

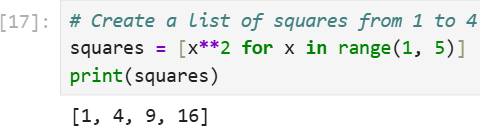
**EXAMPLE:**

****

**List Comprehension in Python:**

List comprehension is a concise way to create lists using a single line of code. It is useful for applying an operation or filter to items in an iterable, such as a list or range.

**EXAMPLE:**

****

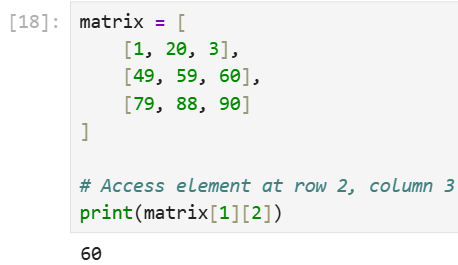
**Explanation:**

* **for x in range(1, 5):** loops through each number from 1 to 4 (excluding 5).
* **x\*\*2:** squares each number x.
* **[ ]:** collects all the squared numbers into a new list.

**Nested Lists in Python:**

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

**EXAMPLE:**

****

**PYTHON TUPLES:**

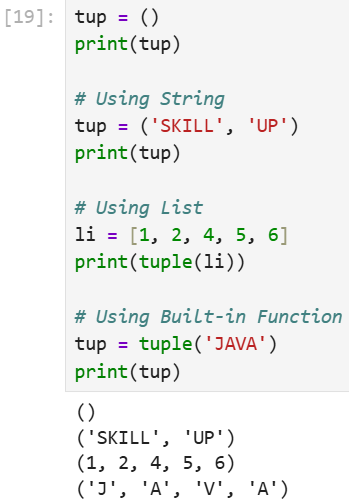
A tuple in Python is **an immutable ordered collection of elements**.

* Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).
* Tuples can hold elements of different data types.
* The main characteristics of tuples are being ordered , heterogeneous and immutable.

**Creating a Tuple:**

A tuple is created by placing all the items inside parentheses (), separated by commas. A tuple can have any number of items and they can be of different data types.

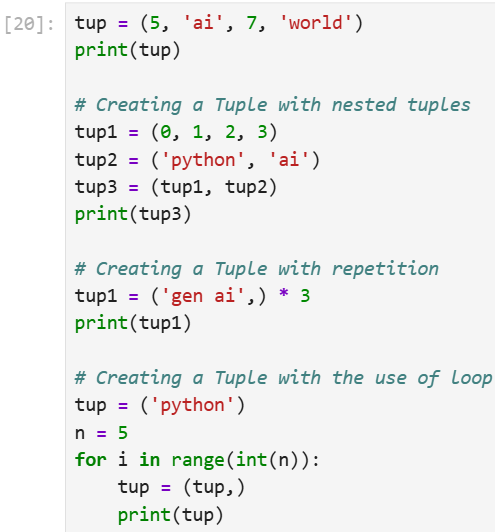
**EXAMPLE:**

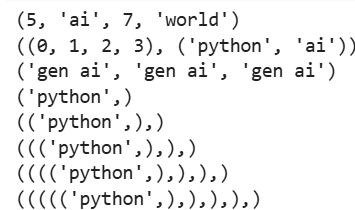
****

**Creating a Tuple with Mixed Datatypes.**

Tuples can contain elements of various data types, including other tuples, lists, dictionaries and even functions.

**EXAMPLE:**

****

****

**Python Tuple Basic Operations**

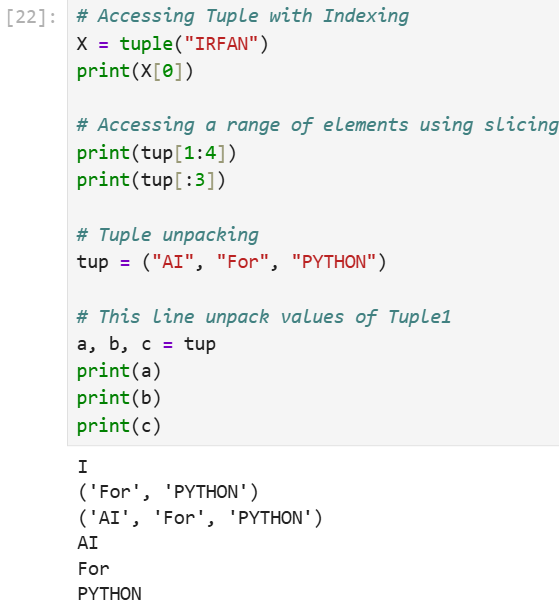
Below are the Python tuple operations.

* Accessing of Python Tuples
* Concatenation of Tuples
* Slicing of Tuple
* Deleting a Tuple

**Accessing of Tuples**

We can access the elements of a tuple by using indexing and slicing, similar to how we access elements in a list. Indexing starts at 0 for the first element and goes up to n-1, where n is the number of elements in the tuple. Negative indexing starts from -1 for the last element and goes backward.

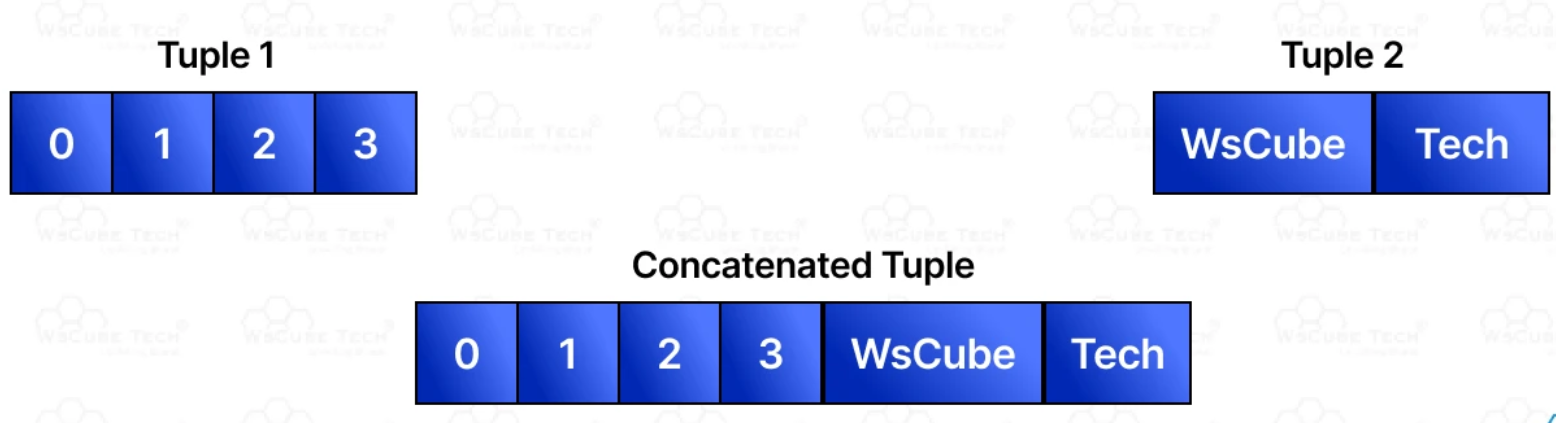
**EXAMPLE:**

****

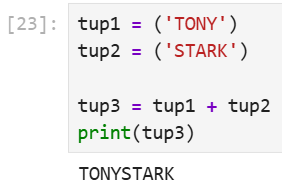
**Concatenation of Tuples**

Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

**Note:** Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.



**EXAMPLE:**

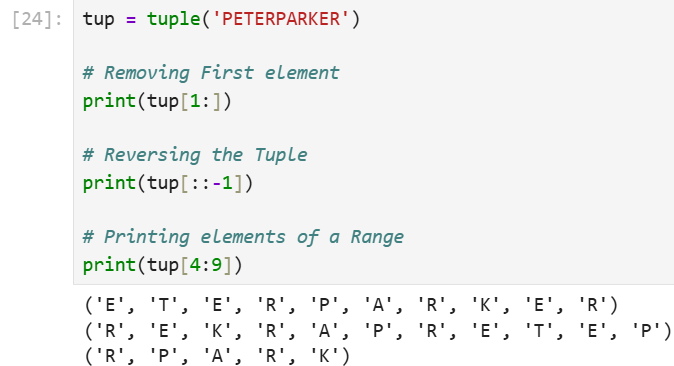
****

**Slicing of Tuple:**

Slicing a tuple means creating a new tuple from a subset of elements of the original tuple. The slicing syntax is tuple[start:stop:step].

**Note-** Negative Increment values can also be used to reverse the sequence of Tuples.

**EXAMPLE:**

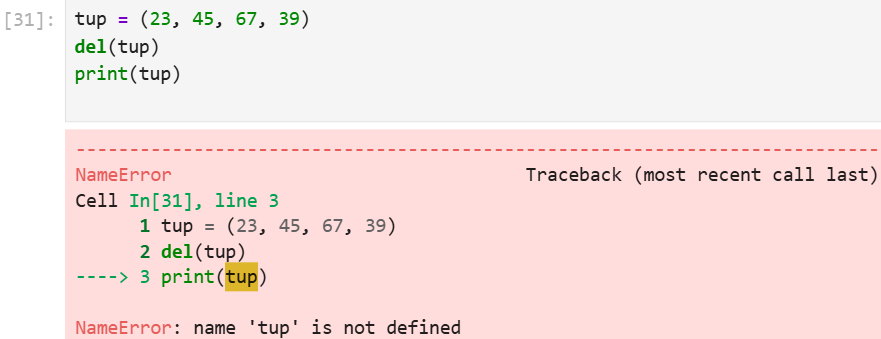
****

**Deleting a Tuple:**

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using del statement.

**Note:** Printing of Tuple after deletion results in an Error.

**EXAMPLE:**

****

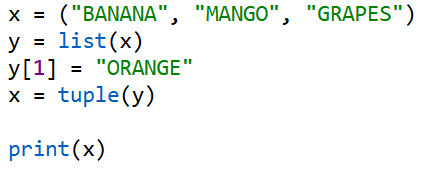
**UPDATE TUPLES:**

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

**Change Tuple Values**

* Once a tuple is created, you cannot change its values. **Tuples are unchangeable, or immutable as it also is called.**
* But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

**EXAMPLE:**

****

**DICTIONARY:**

**Python dictionary**is a data structure that stores the value in **key: value** pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be **immutable**.

A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

**NOTE:** Dictionaries are written with curly brackets, and have keys and values:

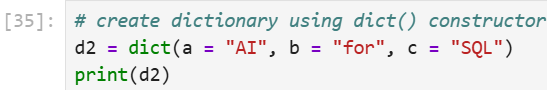
**EXAMPLE:**

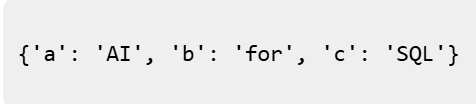


**How to Create a Dictionary**

Dictionary can be created by placing a sequence of elements within **curly {} braces, separated by a 'comma'.**

**EXAMPLE:**

****

****

* **Dictionary keys are case sensitive:** the same name but different cases of Key will be treated distinctly.
* **Keys must be immutable:** This means keys can be strings, numbers or tuples but not lists.
* **Keys must be unique:** Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
* Dictionary internally uses Hashing. Hence, operations like search, insert, delete can be performed in Constant Time.

**Accessing Dictionary Items**

We can access a value from a dictionary by using the **key**within square brackets or **g**et() method.

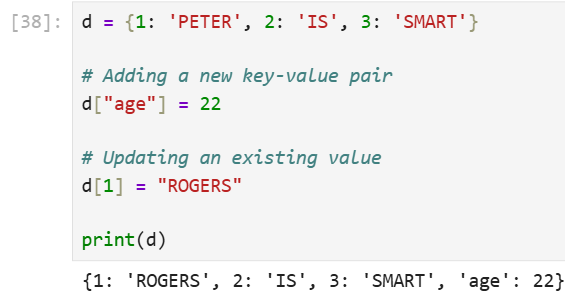
**EXAMPLE:**

****

**Adding and Updating Dictionary Items:**

We can add new key-value pairs or update existing keys by using assignment.

**EXAMPLE:**

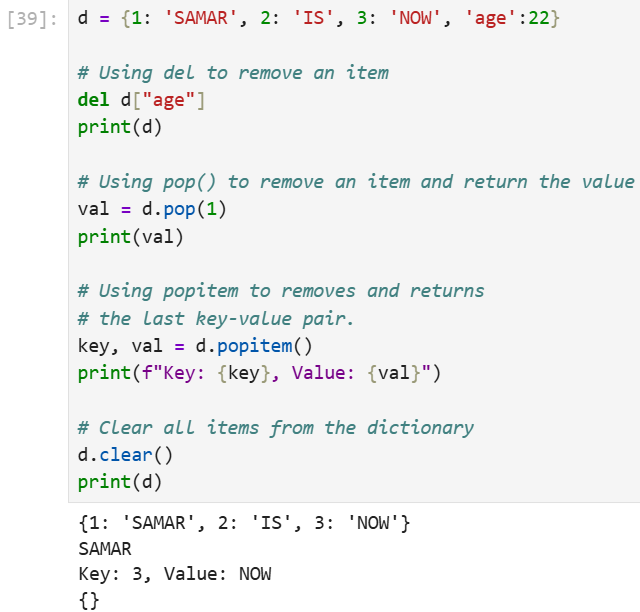
****

**Removing Dictionary Items:**

We can remove items from dictionary using the following methods:

* **del**: Removes an item by key.
* **pop():** Removes an item by key and returns its value.
* **clear():** Empties the dictionary.
* **popitem():** Removes and returns the last key-value pair.

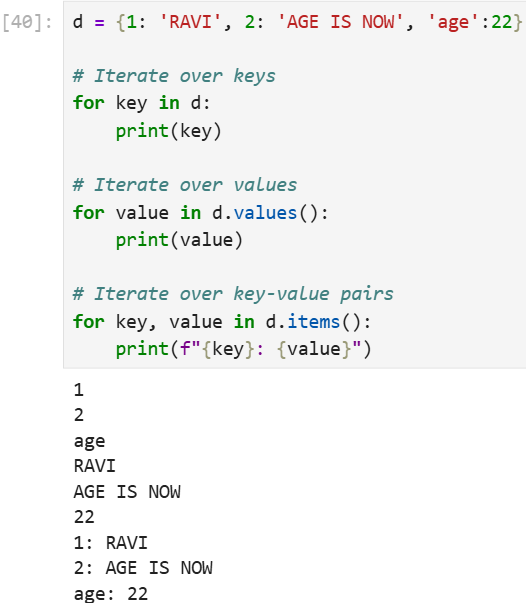
**EXAMPLE:**

****

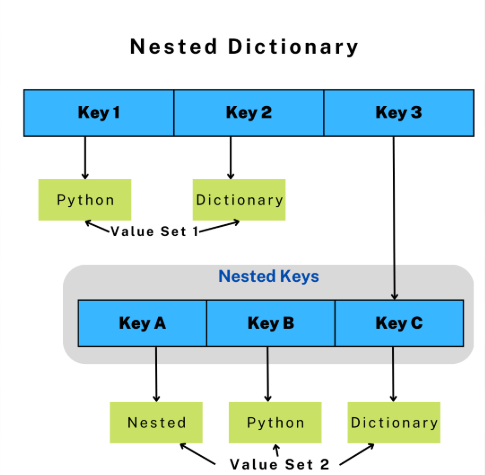
**Iterating Through a Dictionary:**

We can iterate over keys [using keys() method] , values [using values() method] or both [using item() method] with a for loop.

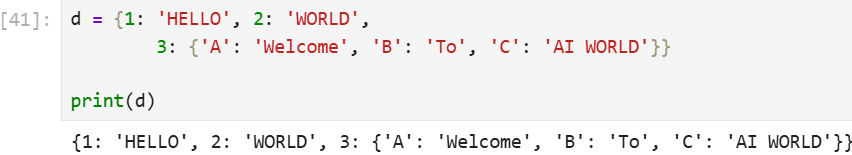
**EXAMPLE:**

****

**NESTED DICTIONARIES IN PYTHON:**



**EXMAPLE FOR NESTED DICTIONARY:**

****

**Copying Dictionaries in Python:**

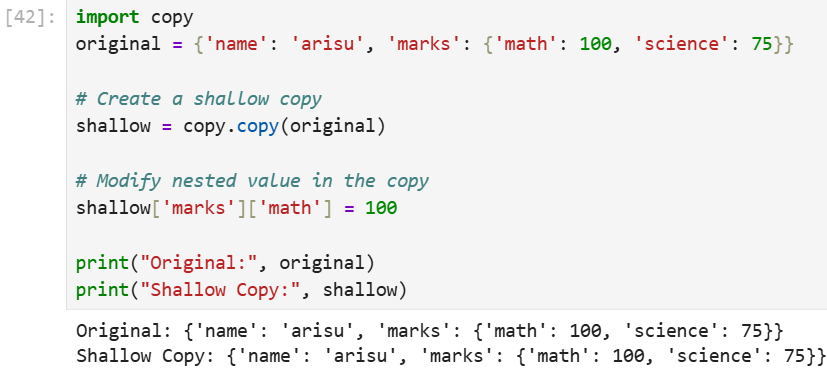
A copy of a dictionary can be created using either shallow copy or deep copy methods. These methods allow duplicating dictionary objects, but they behave differently when it comes to nested data. Let's discuss both in detail.

1. **Shallow Copy**

A shallow copy makes a new dictionary with same outer values as the original. But if the dictionary has nested data (like a list or another dictionary), both copies still share that inner data. So, changes to nested parts will affect other.

It is created using copy.copy() method from Python’s copy module.

**EXAMPLE:**

****

**Explanation:**

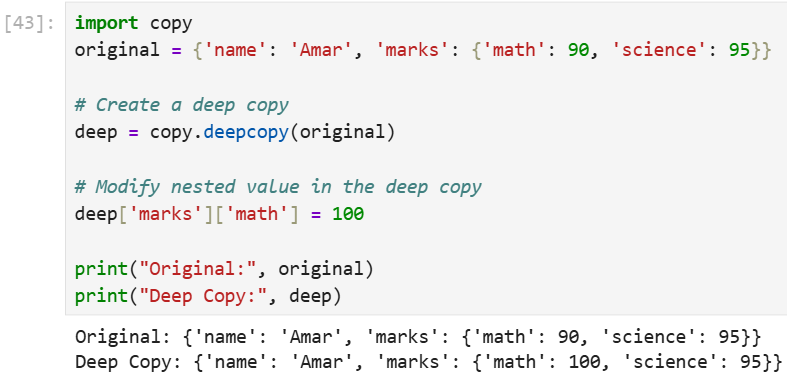
* **shallow = copy.copy(original):**creates a shallow copy, nested '**marks**' remains shared.
* **shallow['marks']['math'] = 100:** updates '**math**' in the shared nested dictionary.
* **print(original), print(shallow):** both show updated '**math**' value due to shared data.

**2. Deep Copy**

A **deep copy** makes a new dictionary and also creates separate copies of all nested data (like lists or other dictionaries). This means original and copy are completely independent, changes made to nested parts do not affect other.

It is created using **copy.deepcopy()** method from Python’s copy module.

**EXAMPLE:**

****

**Explanation:**

* **deep = copy.deepcopy(original):** creates a deep copy, nested '**marks**' is also copied separately.
* **deep['marks']['math'] = 100:** updates '**math**' in the deep copy’s nested dictionary only.
* **print(original), print(deep):** original remains unchanged, only deep copy shows the updated '**math**' value.

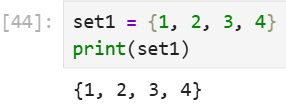
**PYTHON SETS:**

Python set is an **unordered collection**of multiple items having different datatypes. In Python, sets are **mutable**, **unindexed**and do not contain duplicates. The order of elements in a set is not preserved and can change.

**Creating a Set in Python**

In Python, the most basic and efficient method for creating a set is using curly braces.

**EXAMPLE:**

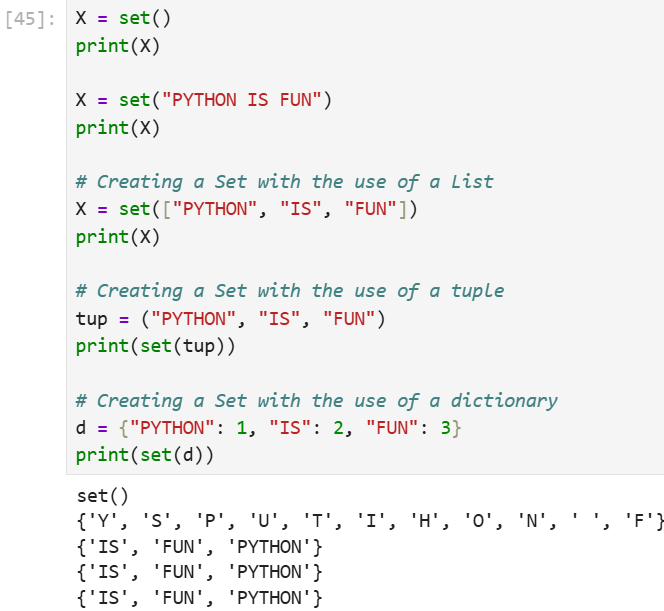
****

**Using the set() function:**

Python Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'.

**Note:** A Python set cannot contain mutable types such as lists or dictionaries, because they are unhashable.

**EXAMPLE:**

****

**Unordered, Unindexed and Mutability:**

In set, the order of elements is not guaranteed to be the same as the order in which they were added. The output could vary each time we run the program. Also the duplicate items entered are removed by itself.

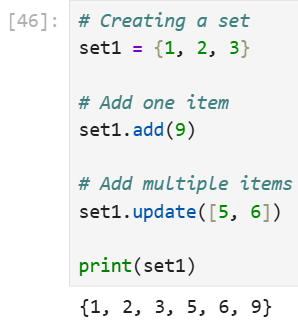
**NOTE:** Sets do not support indexing. Trying to access an element by index (set[0]) raises a TypeError.

We can add elements to the set using **add()**. We can remove elements from the set using **remove()**. The set changes after these operations, demonstrating its mutability. However, we cannot changes its items directly.

**Adding Elements to a Set in Python:**

We can add items to a set using add() method and update() method. add() method can be used to add only a single item. To add multiple items we use update() method.

**EXAMPLE:**

****

**Removing Elements from the Set in Python:**

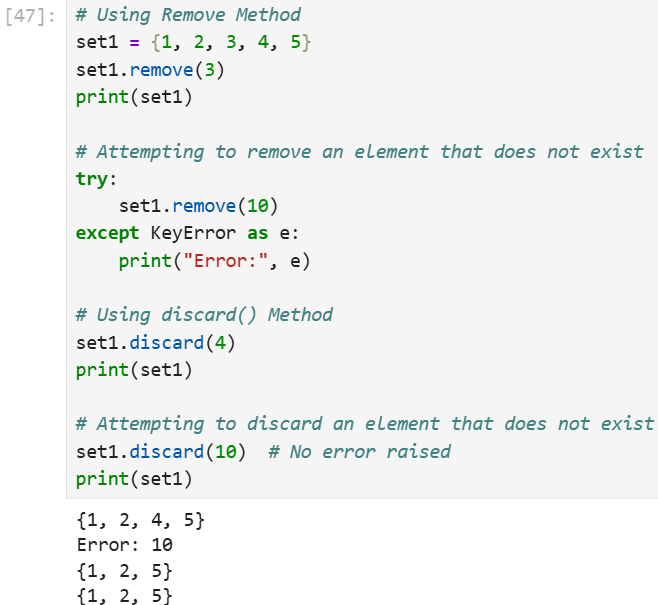
We can remove an element from a set in Python using several methods: remove(), discard() and pop(). Each method works slightly differently:

* **Using remove() Method or discard() Method**
* **Using pop() Method**
* **Using clear() Method**

Using remove() Method or discard() Method

remove() method removes a specified element from the set. If the element is not present in the set, it raises a Key Error. discard() method also removes a specified element from the set. Unlike remove(), if the element is not found, it does not raise an error.

**EXAMPLE:**

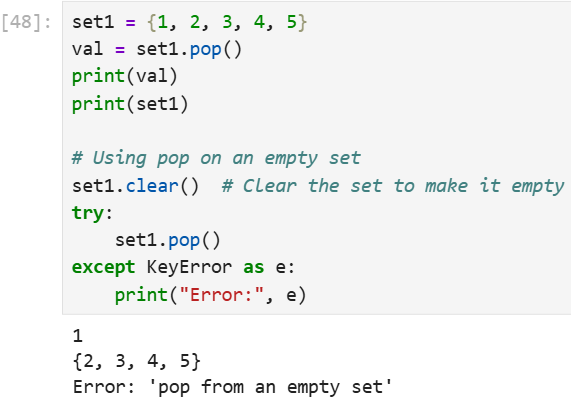
****

**Using pop() Method:**

pop() method removes and returns an arbitrary element from the set. This means we don't know which element will be removed. If the set is empty, it raises a KeyError.

**Note:** If the set is unordered then there's no such way to determine which element is popped by using the pop() function.

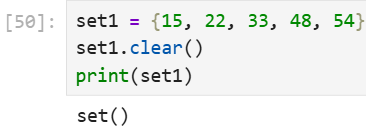
**EXAMPLE:**

****

**Using clear() Method:**

clear() method removes all elements from the set, leaving it empty.

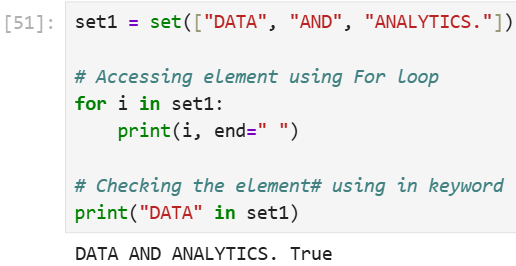
**EXAMPLE:**

****

**Accessing a Set in Python:**

We can loop through a set to access set items as set is unindexed and do not support accessing elements by indexing. Also we can use in keyword which is membership operator to check if an item exists in a set.

**EXAMPLE:**

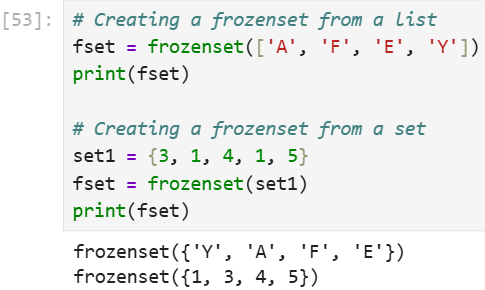
****

**Frozen Sets in Python:**

A frozenset in Python is a built-in data type that is similar to a set but with one key difference that is immutability. This means that once a frozenset is created, we cannot modify its elements that is we cannot add, remove or change any items in it. Like regular sets, a frozenset cannot contain duplicate elements.

**NOTE:** If no parameters are passed, it returns an empty frozenset.

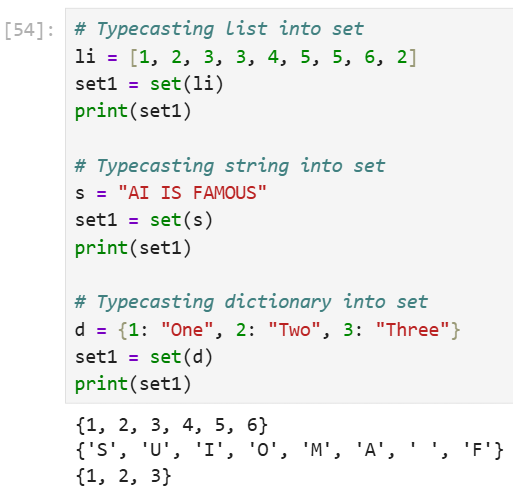
**EXAMPLE:**

****

**Typecasting Objects into Sets:**

Typecasting objects into sets in Python refers to converting various data types into a set. Python provides the set() constructor to perform this typecasting, allowing us to convert lists, tuples and strings into sets.

**EXAMPLE:**

****

**CONCLUSION:**

Overall, sets can be a useful data structure in Python, especially for removing duplicates or for fast membership testing. However, their lack of ordering and limited functionality can also make them less versatile than lists or dictionaries, so it is important to carefully consider the advantages and disadvantages of using sets when deciding which data structure to use in your Python program.

**PYTHON FUNCTIONS:**

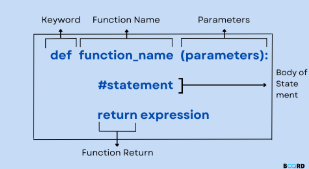
is a block of statements that does a specific task. The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

**Benefits of Using Functions**

* Code Reuse
* Reduced code length
* Increased readability of code

**Python Function Declaration:**

The syntax to declare a function is:



**Types of Functions in Python:**

Below are the different types of functions in Python:

* Built-in library function: These are Standard functions in Python that are available to use.
* User-defined function: We can create our own functions based on our requirements.

**Creating a Function in Python:**

We can define a function in Python, using the def keyword. We can add any type of functionalities and properties to it as we require.

**What is def?**

The def keyword stands for define. It is used to create a user-defined function. It marks the beginning of a function block and allows you to group a set of statements so they can be reused when the function is called.

**Syntax:**

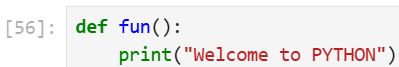
def function\_name(parameters):

# function body

**Explanation:**

* **def:**Starts the function definition.
* **function\_name:** Name of the function.
* **parameters:** Inputs passed to the function (inside ()), optional.
* **:** : Indicates the start of the function body.
* **Indented code:**The function body that runs when called.

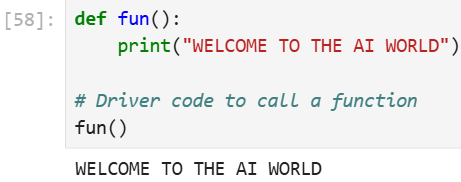
**EXAMPLE:**

****

**Calling a Function in Python**

After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function. Below is the example for calling def function Python.

**EXAMPLE:**

****

**Python Function Arguments:**

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

**Syntax for functions with arguments:**

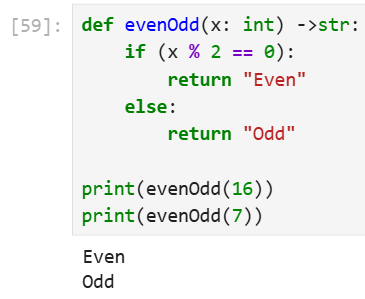
def function\_name(parameter: data\_type) -> return\_type:  
"""Docstring"""  
# body of the function  
return expression

**data\_type and return\_type are optional in function declaration, meaning the same function can also be written as:**

def function\_name(parameter) :  
"""Docstring"""  
# body of the function  
return expression

Let's understand this with an example, we will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.

**EXAMPLE:**

****

**Types of Python Function Arguments:**

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following function argument types in Python:

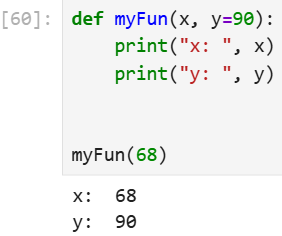
* **Default argument**
* **Keyword arguments (named arguments)**
* **Positional arguments**
* **Arbitrary arguments** (variable-length arguments \*args and \*\*kwargs)

Let's discuss each type in detail.

**Default Arguments:**

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments to write functions in Python.

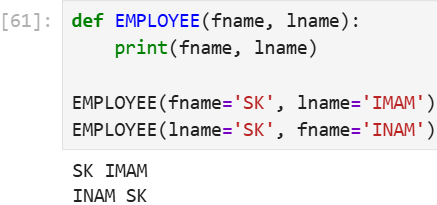
**EXAMPLE:**

****

**Keyword Arguments:**

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

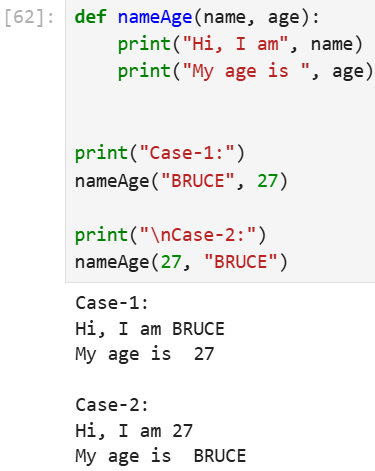
**EXAMPLE:**

****

**Positional Arguments:**

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

**EXAMPLE:**

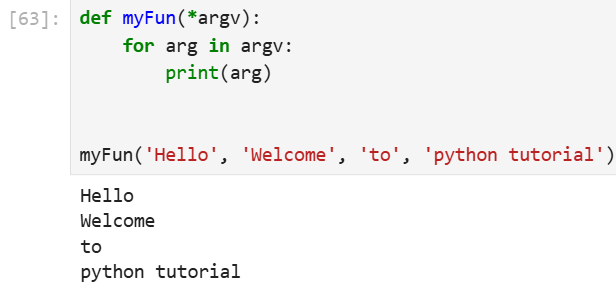
****

**Arbitrary Keyword Arguments:**

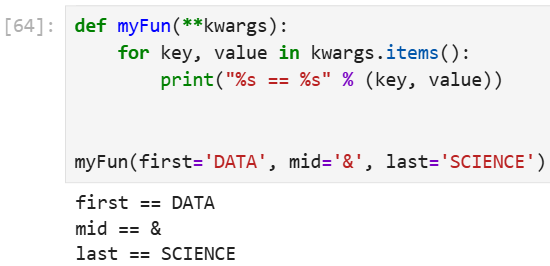
In Python Arbitrary Keyword Arguments, \*args, and \*\*kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

* \*args in Python (Non-Keyword Arguments)
* \*\*kwargs in Python (Keyword Arguments)

**EXAMPLE 1:**

****

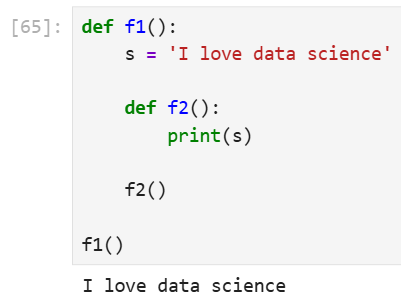
**EXAMPLE 2:**

****

**Python Function within Functions:**

A function that is defined inside another function is known as the **inner function** or **nested function**. Nested functions can access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

**EXAMPLE:**

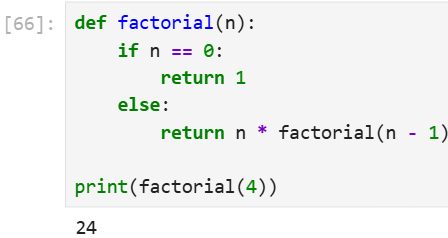
****

**RECURSIVE FUNCTIONS IN PYTHON:**

**Recursion**in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve **Mathematical and Recursive Problems.**

Using a recursive function should be done with caution, as a recursive function can become like a non-terminating loop. It is better to check your exit statement while creating a recursive function.

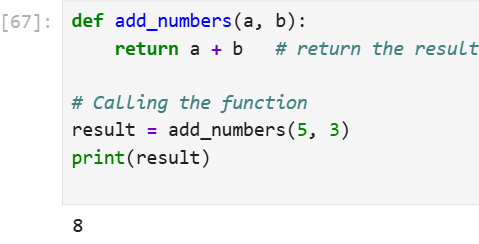
**EXAMPLE:**

****

**Return Statement in Python Function:**

The **return**statement in Python is used to exit a function and send a value back to the caller. It can return any **data type**, and if multiple values are separated by commas, they are automatically packed into a **tuple**. If no value is specified, the function returns **None**by default.

**EXAMPLE:**

****

**Syntax:**

return [expression]

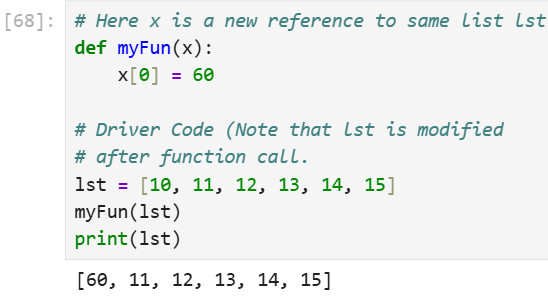
**Explanation:**

* return: Ends the function and optionally sends a value to the caller.
* [expression]: Optional value to return, defaults to None if omitted.

**Pass by Reference and Pass by Value**

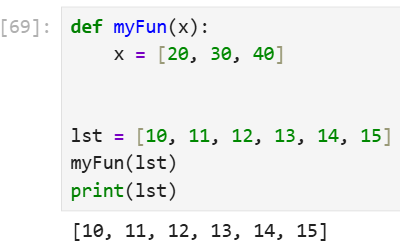
One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function Python, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

**EXAMPLE:**

****

When we pass a reference and change the received reference to something else, the connection between the passed and received parameters is broken. For example, consider the below program as follows.

**EXAMPLE:**

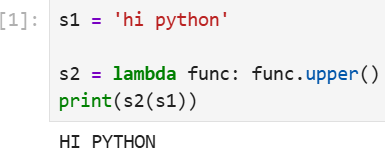
****

**LAMBDA FUNCTIONS:**

**Python Lambda Functions** are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python.

**EXAMPLE:**

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using upper ().



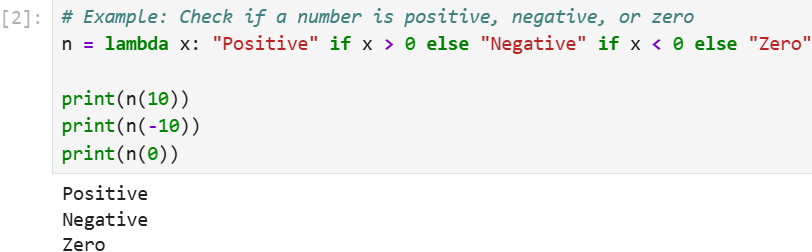
This code defines a lambda function named s2 that takes a string as its argument and converts it to uppercase using the upper() method. It then applies this lambda function to the string 'hi python' and prints the result.

**Syntax:** lambda arguments : expression

**lambda with Condition Checking:**

A lambda function can include conditions using if statements

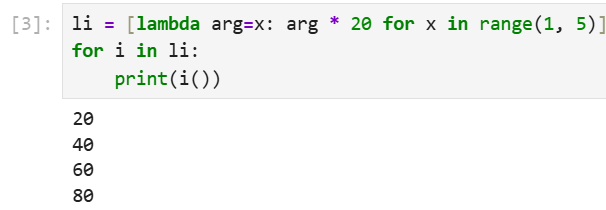
**EXAMPLE:**

****

**Lambda with List Comprehension:**

Combining lambda with list comprehensions enables us to apply transformations to data in a concise way.

**EXAMPLE:**

****

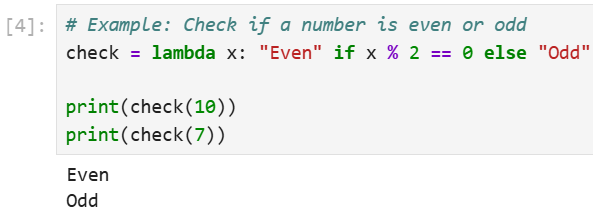
**EXPLANATION:**

* The lambda function squares each element.
* The list comprehension iterates through li and applies the lambda to each element.
* This is ideal for applying transformations to datasets in data preprocessing or manipulation tasks.

**Lambda with if-else:**

lambda functions can incorporate conditional logic directly, allowing us to handle simple decision making within the function.

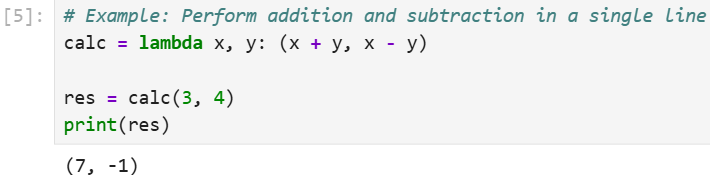
**EXAMPLE:**

****

**Lambda with Multiple Statements:**

Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function.

**EXAMPLE:**

****

**EXPLANATION:**

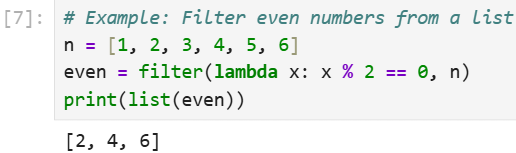
* The lambda function performs both addition and multiplication and returns a tuple with both results.
* This is useful for scenarios where multiple calculations need to be performed and returned together.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

**Using lambda with filter()**

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True.

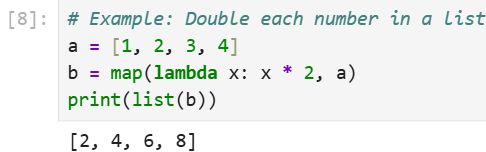
**EXAMPLE:**

****

**Using lambda with map():**

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a new list is returned which contains all the lambda-modified items returned by that function for each item.

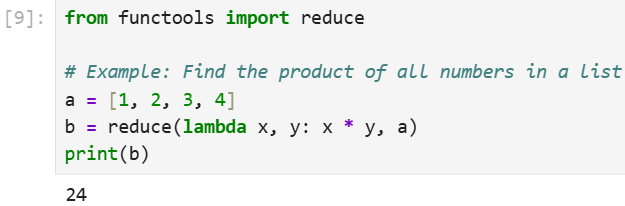
**EXAMPLE:**

****

**Using lambda with reduce():**

The reduce() function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the **functools**module.

**EXAMPLE:**

****

**RECURSIONS IN PYTHON:**

Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem by breaking it into smaller, simpler subproblems.

In Python, recursion is especially useful for problems that can be divided into identical smaller tasks, such as mathematical calculations, tree traversals or divide-and-conquer algorithms.

**Working of Recursion:**

A **recursive function** is just like any other Python function except that it calls itself in its body. **Let's see basic structure of recursive function:**

def recursive\_function(parameters):  
if base\_case\_condition:  
return base\_result  
else:  
return recursive\_function(modified\_parameters)

**Recursive function** contains two key parts:

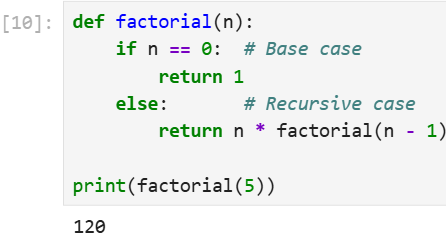
* **Base Case:** The stopping condition that prevents infinite recursion.
* **Recursive Case:** The part of the function where it calls itself with modified parameters.

**Examples of Recursion**

Let's understand recursion better with the help of some examples.

**Example 1: Factorial Calculation**

This code defines a recursive function to calculate factorial of a number, where function repeatedly calls itself with smaller values until it reaches the base case.

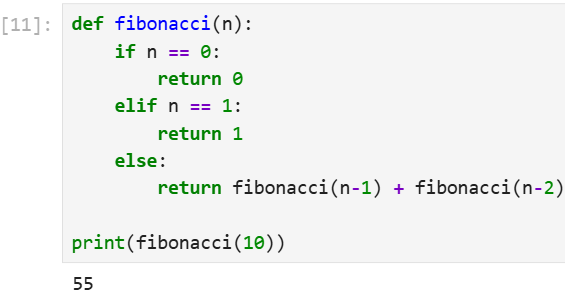


**EXPLANATION:**

* **Base Case:** When **n == 0**, recursion stops and returns **1**.
* **Recursive Case:** Multiplies **n**with the factorial of **n-1** until it reaches the base case.

**Example 2: Fibonacci Sequence**

This code defines a recursive function to calculate nth Fibonacci number, where each number is the sum of the two preceding ones, starting from 0 and 1.



**EXPLANATION:**

* **Base Cases:**If n == 0, the function returns 0. If n == 1, the function returns 1. These two cases are necessary to stop the recursion.
* **Recursive Case:**function calls itself twice with decrements of n (i.e., fibonacci(n-1) and fibonacci(n-2)), summing results of these calls.

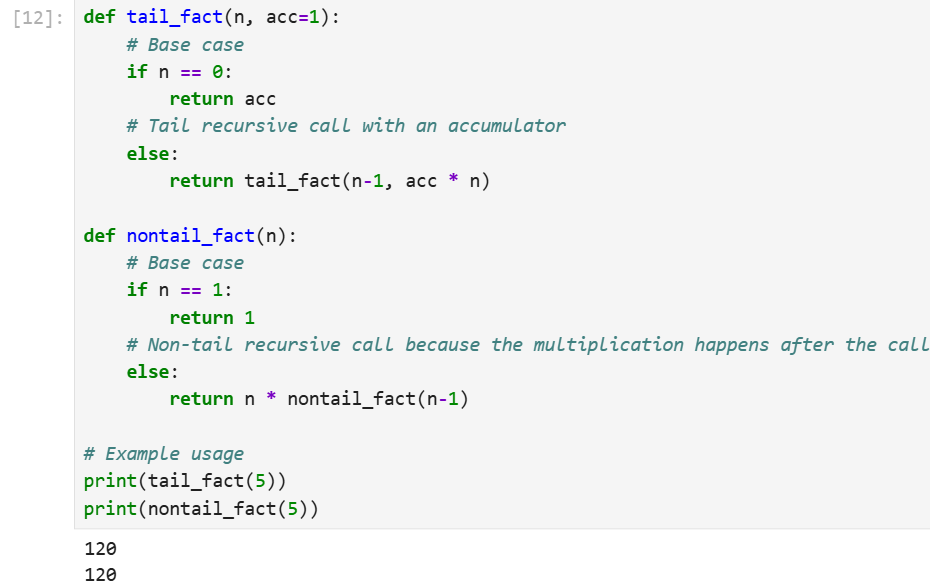
**Types of Recursion in Python:**

Recursion can be broadly classified into two types: **tail recursion** and **non-tail recursion**. The main difference between them is related to what happens after recursive call.

* **Tail Recursion:** The recursive call is the last thing the function does, so nothing happens after it returns. Some languages can optimize this to work like a loop, saving memory.
* **Non-Tail Recursion:** The function does more work after the recursive call returns, so it can’t be optimized into a loop.

**Example:**

This code compares tail recursion and non-tail recursion using two versions of factorial function one with an accumulator (tail-recursive) and one with multiplication after recursive call (non-tail-recursive).



**EXPLANATION:**

* **def tail\_fact(n, acc=1):** - Defines a tail-recursive factorial function with an accumulator acc to store intermediate results.
* **if n == 0: return acc** - Base case: when n reaches 0, return the accumulated result.
* **return tail\_fact(n-1, acc \* n)** - Tail-recursive call: multiplies acc by n before the call, so no extra work is left after recursion.
* **def nontail\_fact(n):** - Defines a non-tail-recursive factorial function.
* **if n == 1: return 1** - Base case: factorial of 1 is 1.
* **return n \* nontail\_fact(n-1)** - Non-tail call: multiplication happens after the recursive call returns, so more work remains after recursion.

**When to Avoid Recursion:**

* When the problem can be solved easily with loops.
* When recursion depth is large enough to risk a stack overflow.
* When performance is critical and function call overhead matters.

**Advantages:**

* **Simplicity:**Recursive code is generally simpler and cleaner, especially for problems inherently recursive in nature (e.g., tree traversals, dynamic programming problems).
* **Reduced Code Length:** Recursion can reduce the length of the code since the repetitive tasks are handled through repeated function calls.

**Disadvantages:**

* **Memory Overhead:** Each recursive call adds a new layer to the stack, which can result in significant memory use, especially for deep recursion.
* **Performance Issues:** Recursive functions may lead to slower responses due to overheads like function calls and returns.
* **Risk of Stack Overflow:** Excessive recursion can lead to a stack overflow error if the recursion depth exceeds the stack limit.

**LIST COMPREHENSIONS IN PYTHON:**

List comprehension is a concise and powerful way to create new lists by applying an expression to each item in an existing iterable (like a list, tuple or range). It helps you write clean, readable and efficient code compared to traditional loops.

**Syntax:**

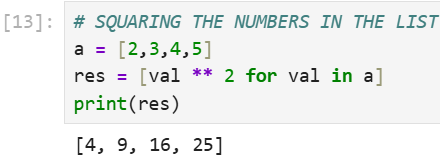
[expression for item in iterable if condition]

**Parameters:**

* **expression:** operation or value to include in the new list.
* **item:** current element from the iterable.
* **iterable:** sequence like a list, tuple or range.
* **if condition (optional):** filter to include only items that satisfy the condition.

**Example:**

Suppose you want to square every number in a list:



**Why Use List Comprehension?**

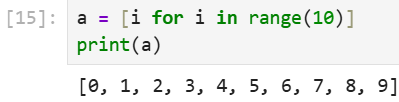
* **Cleaner code:** Combines looping, filtering and transformation in one line.
* **More readable:** Avoids verbose loops and temporary variables.
* **Faster execution:** Often faster than equivalent for-loops.

**EXAMPLES OF LIST COMPREHENSION:**

1. **Creating a list from a range**

One can quickly create a list of numbers within a specific range using list comprehension. This example generates numbers from 0 to 9 and stores them in a list.

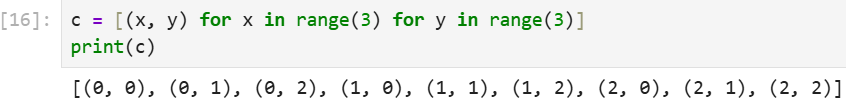
**EXAMPLE:**



1. **Using nested loops**

A list of all coordinate pairs in a 3x3 grid can be generated by combining two loops inside a list comprehension. This example produces every possible (x, y) pair where both x and y range from 0 to 2.

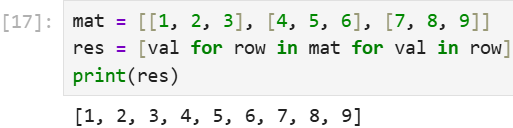
**EXAMPLE:**

****

1. **Flattening a list of lists**

A nested list (matrix) can be transformed into a single flat list by iterating through each sublist and its elements. This example flattens a 3x3 matrix into one continuous list of values.

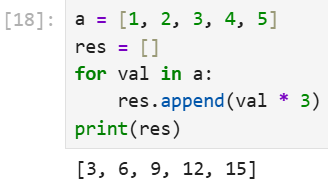
**EXAMPLE:**

****

**For Loop vs. List Comprehension:**

A for loop takes multiple lines to build a new list by iterating and appending each item manually. List comprehension does same in just one line, making code shorter and easier to read.

**EXAMPLE:**

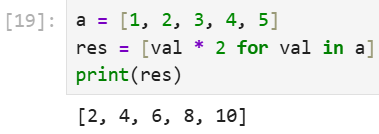
****

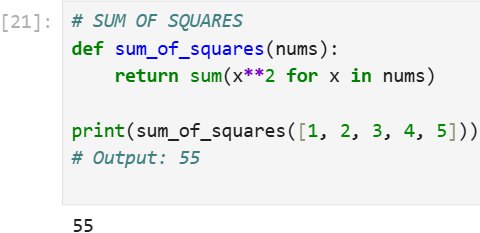
**EXPLANATION:**

* **res = []**creates an empty list to store results.
* **for val in a:** loops through each number in list **a**.
* **res.append(val \* 2)** doubles current number val and adds it to **res**list.

**USING LIST COMPREHENSION:**

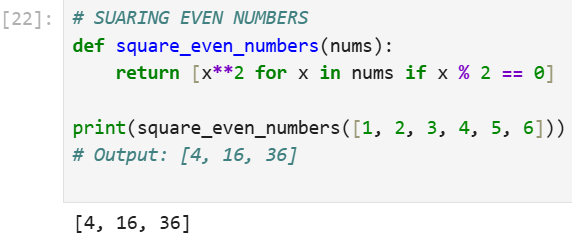
**EXAMPLE 2:**

****

****

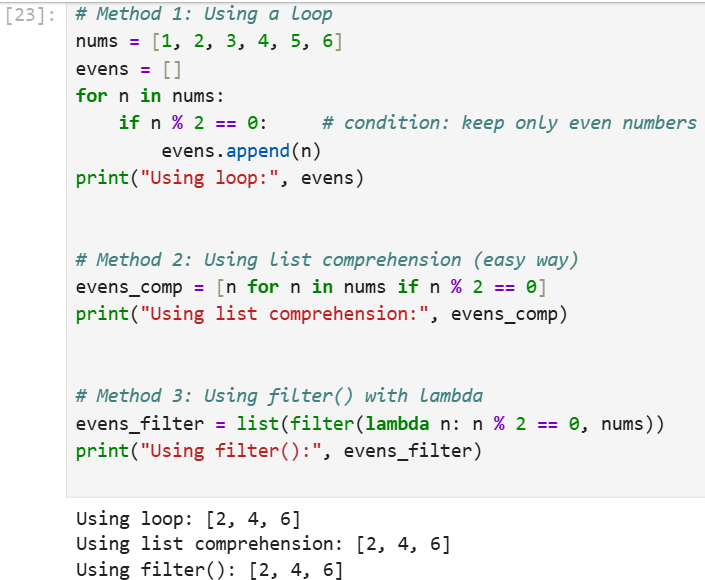
**EXPLANATION:**

* x\*\*2 → squares each number.
* for x in nums → loops through the list.
* sum(...) → adds all squared values together.  
  So, 1² + 2² + 3² + 4² + 5² = 55.



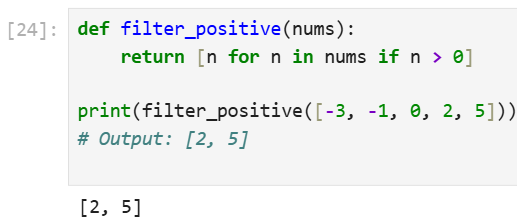
**EXPLANATION:**

* **if x % 2 == 0** → selects only even numbers.
* **[x\*\*2 ...]** → creates a new list with squared values.



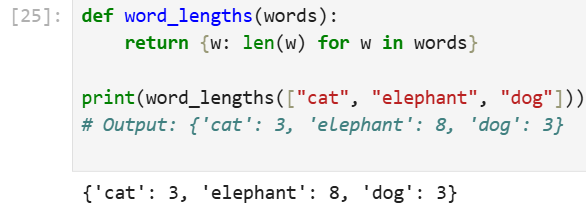
**EXPLANATION:**

* **Loop** → manually check each number and append if it meets condition.
* **List comprehension** → shorter, Pythonic way of writing the same logic.
* **filter()** → uses a function (lambda) to keep only items that return True.



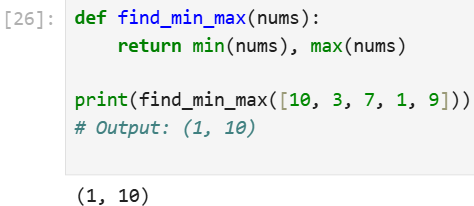
**EXPLANATION:**

* **for n in nums** → goes through each number in the list.
* **if n > 0** → keeps only numbers greater than 0.
* **[n ...]** → builds a new list with those numbers.
* **Result:** from [-3, -1, 0, 2, 5] only 2 and 5 pass the test → [2, 5].

****

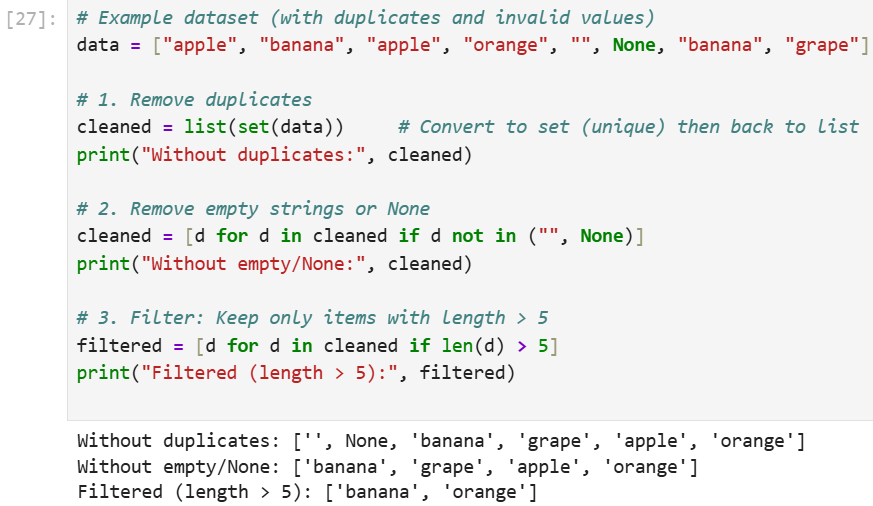
**EXPLANATION:**

* **for w in words** → goes through each word in the list.
* **len(w)** → finds the length of that word.
* **{w: len(w) ...}** → creates a dictionary where each **word is the key** and its **length is the value**.
* **Example:**
* "cat" → length 3
* "elephant" → length 8
* "dog" → length 3
* Result: {'cat': 3, 'elephant': 8, 'dog': 3}



**EXPLANATION:**

* **min(nums)** → finds the smallest number in the list.
* **max(nums)** → finds the largest number in the list.
* The function returns both as a **tuple** (min\_value, max\_value).
* For [10, 3, 7, 1, 9]:
* Smallest → 1
* Largest → 10
* **Result:** (1, 10)



**EXPLANATION:**

* **Remove duplicates** → set() automatically keeps only unique values.
* **Remove invalid values** → list comprehension filters out "" (empty) and None.
* **Filter data** → condition len(d) > 5 keeps only longer words.