Functions

Functions help in encapsulating logic and prevent us from using the same piece of code again and again.

Description	Syntax	Example

Defining a function in python	def name_of_function():
	  ```Function body```
	
	# Calling the function
	name_of_function()
		def make_tea():
		  print("I am making tea")
		make_tea()
		> I am making tea

Functions can take in parameters that can be used within a string body.	def name_of_function(param):
	  ```Function body```
		def print_name(name):
		  print("My name is", name)
		print_name("Rob")
		> My name is Rob

The "return" keyword from within the function can be used to return a value after a function call.	def name_of_function(x, y):
	  # do something
	  return z
		def return_sum(x, y):
		  z = x + y
		  return z
		my_sum = return_sum(1, 2)
		print(my_sum)
		> 3

Type of arguments in Functions

Positional Arguments	def random(a, b, c, d): print(b, c, a, d) random(4, 5, 6, 7) > 5 6 4 7
Keyworded Arguments	def random(a, b, c, d): print(b, c, a, d) random(a = 4, b = 5, c = 6, d = 7) > 4 5 6 7
All keyword arguments **must** follow all positional arguments	**Wrong Syntax** random(b = 4, 5, 6, 7) > Error **Correct Syntax** random(4, 5, c = 6, d = 7) > 4 5 6 7
Scope of variables	**Local Variable inside the function is present** a = 10 # Global Variable def random(): a = 50 # Local Variable

	print(a) random() > 50 **Local Variable inside the function is not present** a = 10 # Global Variable def random(): print(a) random() > 10
“global” keyword in front of a variable name is used to tell python to use the global value of that variable.	a = 10 def random(): global a a = 20 print(a) random() print(a) > 20 > 20

Lists

Creating a list	list_name = [element_1, element_2, …] **Example** numbers = [1, 2, 3, 4, 5] print(numbers) > [1, 2, 3, 4, 5]

Accessing elements from a list

Description	**Syntax**	**Example**
Indexing a list to access a value. The index number ranges between 0 and the length of the list.	list_name[index_number]	nums = [1, 2, 3] print(nums[0]) > 1 print(nums[2]) > 3 print(nums[3]) > IndexError

Negative indexing to access elements from the end.	list_name[index_number]	nums = [1, 2, 3] print(nums[-1]) > 3

Methods to add values to a list

Description	Syntax	Example
list.append(): Adds an element at the end of a list.	list_name.append(element)	nums = [1, 2, 3] nums.append(4) print(nums) > [1, 2, 3, 4]
list.insert(): Add an element at a specific index in the list.	list_name.insert(index_number, element_to_insert)	nums = [1, 3, 4] nums.insert(1, 2) print(nums) > [1, 2, 3, 4]
list.extend(): Add multiple values at the end of a list.	list_name.extend(list_of_values)	nums = [1] nums.extend([2, 3, 4]) print(nums) > [1, 2, 3, 4]

List Slicing

- Used for accessing a range of elements from a list.

Description	Syntax	Example
Accessing a range of elements	list_name[start_idx : end_idx]	nums = [1, 2, 3, 4, 5] slice = nums[1:3] print(slice) > [2, 3]
Accessing values at odd indexes.	list_name[start:end:step_size]	nums = [1, 2, 3, 4, 5] slice = nums[0:len(nums):2] print(slice) > [1, 3, 5]
Accessing all elements from a particular index till the end.	list_name[start:]	nums = [1, 2, 3, 4, 5] slice = nums[2:] print(slice) > [3, 4, 5]
Accessing all the elements from a list from a particular index till the start of the list.	list_name[:end]	nums = [1, 2, 3, 4, 5] slice = nums[:3] print(slice) > [1, 2, 3]

A step size of -1 can be used to traverse a list from the back. (reversed list)	list_name[::-1]	po

Iterating over a list

Description	Syntax	Example

Iterating using a for loop with the range function.	for i in range(len(list_name)):
	element = list_name[i]
	```do something```
		nums = [1, 2, 3, 4, 5]
		for i in range(len(nums)):
		print(nums[i], end = “ ”)
		> 1 2 3 4 5

Iterating using a for loop without range.	for x in list_name: ```do something```	nums = [1, 2, 3, 4, 5] for x in nums: print(x, end = “ ”) > 1 2 3 4 5

Few more list methods

Description	Syntax	Example
• **list.pop()** → Removes an element at a particular index in a list. • If value no index passed → Last value is popped.	list_name.pop(index_to_pop)	nums = [1, 2, 3] nums.pop() print(nums) > [1, 2] nums.pop(0) print(nums) > [2]
• **list.remove()** → Removes a particular value from the list. • Multiple occurrences → removes the first occurrence • If value not present → Error.	list_name.remove(value_to_remove)	nums = [1, 2, 3] nums.remove(3) print(nums) > [1, 3]
• **list.index()** → Returns the first index of a value in the list that is passed in as a parameter to the	list_name.index(value)	nums = [1, 2, 3] idx =nums.index(2) print(idx) > 1

method. ● Multiple occurrences → removes the first occurrence ● If the value is not present → Error.		
● **list.count()** → Returns the number of times a value passed to the method occurs in the list. ● If value not present → Returns 0	list_name.count(value)	nums = [1, 2, 2, 3] c = nums.count(2) print(c) > 2

2D Lists

Description	**Syntax**	**Example**
Initializing a 2D list	list_name = [sub_lists1, sublist2, …]	mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]
Indexing a 2D list. We need two index values for indexing, one for the outer list and one for the inner list.	list_name[outer_idx][inner_idx]	mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9] print(mat[1][2]) > 6 print(mat[0][0]) > 1

Strings

Strings are a sequence of characters

Description	**Syntax**	**Example**
Defining a string in python	string_name = “string_content”	my_name = “Rob” print(my_name) > Rob
Concatenating two strings → The “+” operator can be used to concatenate(join) two strings.	New_str = string_1 + string_2	first_name = “Rob” last_name = “Meyer” print(first_name + “ ” + last_name) > Rob Meyer

• Strings are compared using lexicographical order (dictionary order) • Note: “A” < “a”	string_1 > string_2 string_1 < string_2 string_1 == string_2	my_name = “Rob” your_name = “Corn” print(my_name > your_name) > True print(my_name < your_name) > False print(my_name == your_name) > False
ord() → Returns ASCII value of a character.	ord(character)	ascii_val = ord(“a”) print(ascii_val) > 97
chr() → Returns the character mapped to the ASCII passed.	chr(ascii_value)	character = chr(23123) print(character) > 裴

String methods

Description	Syntax	Example
str.split() → Splits a string based on a certain separator	string_name.split(separator)	ex = “this#is#a#string#” splitted = ex.split(“#”) print(splitted) > [“this”, “is”, “a”, “string”]
str.join() → Joins a list of strings based on a separator.	separator_string.join(list_of_strings)	print(“ ”.join([“list”, “of”, “strings”])) > list of strings
str.replace() → Replaces a certain occurrence of the first character with a second character.	string_name.replace(char1, char2)	newStr = “this is str”.replace(“ ”, “_”) print(newStr) > this_is_str
• **str.find()** → Finds the exact sequence or substring in the original string and returns the starting index of the substring. • If sequence not present → Returns -1	string_name.find(sequence)	s = “this is str” print(s.find(“is”)) > 2
str.count() → Counts the number of times a character or substring is present within a	string_name.count(sequence)	s = “this is str” print(s.find(“t”)) > 2

string.		

Tuples

A tuple is like an immutable list

Description	Syntax	Example
Declaring a tuple	tuple_name = (elem1, elem2,…)	t = (1, 2, 3, 4)
Elements of a tuple are immutable.	-	t = (1, 2, 3, 4) t[0] = 2 > Error
Positive and Negative indexing including slicing is permitted on tuples.	tuple_name[start:end]	t = (1, 2, 3, 4) print(t[2:8]) > (3, 4) print(t[-2:-8]) > ()
Declaring a tuple with a single element.	tuple_name = (elem1,)	t = (1) print(t) > 1 t = (1,) print(t) > (1,)

Sets

- Sets are unique collections of elements
- We represent a set using curly braces { }
- Sets can store strings, tuples, and booleans inside them at once.
- Sets cannot store Lists, Sets, and dictionaries within them.
- We cannot index elements from a set because they are not ordered.

Description	Syntax	Example
Declaring a set	set_name = {elem1, elem2,...}	s = {1, 2, 2, 3, 4} print(s) > {1, 2, 3, 4}
Declaring an empty set	set_name = set()	s = set() print(s) > set()

Set Methods

Description	Syntax	Example
Adding an element to a set.	set_name.add(element)	s = {1, 2, 2, 3, 4} print(s.add(5)) > {1, 2, 3, 4, 5}
Removing an element from a set.	set_name.remove(element)	s = {1, 2, 2, 3, 4} print(s.remove(2)) > {1, 3, 4}
Popping a random element from a set.	set_name.pop()	s = {1, 2, 2, 3, 4} print(s.pop()) > 1
Updating multiple elements in a set.	set_name.update(elements_to_update)	s = {1} s.update((2, 3, 4, 4, 5)) print(s) > {1, 2, 3, 4, 5}

Set Operations
Add diagram

Description	Syntax	Example
Intersection: Contains elements common to both sets	set1.intersection(set2) or set1 & set2	s1 = {2, 3, 4, 5} s2 = {1, 3, 4, 6, 7, 8} print(s1 & s2) > {3, 4}
Union: Contains all elements in both sets	set1.union(set2) or s1 | s2	s1 = {2, 3, 4, 5} s2 = {1, 3, 4, 6, 7, 8} print(s1 | s2) > {1, 2, 3, 4, 5, 6, 7, 8}
Difference: Contains all the elements in set1 which are not in set2	set1.difference(set2) or s1 - s2	s1 = {2, 3, 4, 5} s2 = {1, 3, 4, 6, 7, 8} print(s1 - s2) > {2, 5}
Symmetric Difference: Contains all the elements in union minus the common elements	set1.symmetric_difference(set2) or s1 ^ s2	s1 = {2, 3, 4, 5} s2 = {1, 3, 4, 6, 7, 8} print(s1 ^ s2) > {1, 2, 5, 6, 7, 8}

Dictionaries

- Dictionaries are key-value pairs

Description	Syntax	Example
Creating dictionaries in python	dict_name = {key1: value1, key2: value2, …}	d = {“Delhi”: 450, “UP”: 700}
Accessing a value from a dictionary	dict_name[key_to_access]	d = {“Delhi”: 450, “UP”: 700} print(d[“UP”]) > 700
• **dict.get()** → Used to access a value from a dictionary. • Advantage: Does not throw an error if the key is not present in the dictionary.	dict_name.get(key_to_access)	d = {“Delhi”: 450, “UP”: 700} print(d.get(“UP”)) > 700 print(d.get(“Haryana”)) > None
Updating a value in a dictionary.	dict_name[key] = new_value	d = {“Delhi”: 450, “UP”: 700} d[“Delhi”] = 500 print(d) > {“Delhi”: 500, “UP”: 700}
Removing a key from the dictionary.	dict_name.pop(key_to_pop)	d = {“Delhi”: 450, “UP”: 700} d.pop(“UP”) print(d) > {“Delhi”: 450}