PYTHON en data SCIeNCE :

Module 1 Summary: Python Basics

Congratulations! You have completed this module. At this point, you know that:

- Python can distinguish among data types such as integers, floats, strings, and Booleans.
- Integers are whole numbers that can be positive or negative.
- Floats include integers as well as decimal numbers between the integers.
- You can convert integers to floats using typecasting, but you cannot convert a float to an integer.
- You can convert integers and floats to strings.
- You can convert an integer or float value to True (1) or False (0).
- Expressions in Python are a combination of values and operations used to produce a single result.
- Expressions perform mathematical operations such as addition, subtraction, multiplication, and so on.
- We use"//" to round off integer divisions, resulting in float values.
- Python follows the order of operations (BODMASS) to perform operations with multiple expressions.
- Variables store and manipulate data, allowing you to access and modify values throughout your code.

- The assignment operator "=" assigns a value to a variable.
- ":" denotes the value of the variable within the code.
- Assigning another value to the same variable overrides the previous value of that variable.
- You can perform mathematical operations on variables using the same or different variables.
- While performing operations with various variables, modifying a value in one variable will lead to changes in the other variables.
- Python string operations involve manipulating text data using tasks such as indexing, concatenation, slicing, and formatting.
- A string is usually written within double quotes or single quotes, including letters, white space, digits, or special characters.
- A string attaches to another variable and is an ordered sequence of characters.
- Characters in a string identify their index numbers, which can be positive or negative.
- We use strings as a sequence to perform sequence operations.
- You can input a stride value to perform slicing while operating on a string.
- Operations like finding the length of the string, combining, concatenating, and replicating, result in a new string.
- You cannot modify an existing string; they are immutable.
- You can perform escape sequences using " " to change the layout of the string.

- In Python, you perform tasks such as searching, modifying, and formatting text data with its pre-built string methods functions.
- You apply a method to a string to change its value, resulting in another string.
- You can perform actions such as changing the case of characters in a string, replacing items in a string, finding items in a string, and so on using pre-built string methods.

-

Python Data Structures Cheat Sheet

List

Package/ Method	Description	Code Example
append()	The `append()` method is used to add an element to the end of a list.	Syntax: 1 1 1 list_name.append(element)

Example:

1. 1

2. 2
3. fruits = ["apple", "banana", "orange"]
4. fruits.append("mango") print(fruits)

copy()	The `copy()` method is used to create a shallow copy of a list.	Example 1: 1 1 2 2 3 3 1 my_list = [1, 2, 3, 4, 5] 2 new_list = my_list.copy() print(new_list) 3 # Output: [1, 2, 3, 4, 5]
count()	The `count()` method is used to count the number of occurrences of a specific element in a list in Python.	Example: 1 1 2 2 3 3 1 my_list = [1, 2, 2, 3, 4, 2, 5, 2] 2 count = my_list.count(2) print(count) 3 # Output: 4

Creating a list	A list is a built-in data type that represents an ordered and mutable collection of elements. Lists are enclosed in square brackets [] and elements are separated by commas.	Example: 1 1 1 fruits = ["apple", "banana", "orange", "mango"]

del	The `del` statement is used to remove an element from list. `del` statement removes the element at the specified index.	Example:
		1 1
		2 2
		3 3
		1 my_list = [10, 20, 30, 40, 50]
		2 del my_list[2] # Removes the element at index 2
		print(my_list)
		3 # Output: [10, 20, 40, 50]

extend()	The `extend()` method is used to add multiple elements to a list. It takes an iterable (such as another list, tuple, or string) and appends each element of the iterable to the original list.	Syntax: 1 1 1 list_name.extend(iterable)

Example:

```
1
2
3
4
fruits = ["apple", "banana", "orange"]
more_fruits = ["mango", "grape"]
fruits.extend(more_fruits)
print(fruits)
```

Indexing	Indexing in a list allows you to access individual elements by their position. In Python, indexing starts from 0 for the first element and goes up to `length_of_list - 1`.	Example: 1 1 2 2 3 3 4 4 5 5 1 my_list = [10, 20, 30, 40, 50] 2 print(my_list[0]) 3 # Output: 10 (accessing the first element) 4 print(my_list[-1]) 5 # Output: 50 (accessing the last element using negative indexing)
insert()	The `insert()` method is used to insert an element.	Syntax: 1 1 1 list_name.insert(index, element)

Example:

1. 1
2. 2
3. 3

4. my_list = [1, 2, 3, 4, 5]
5. my_list.insert(2, 6)
6. print(my_list)

Modifying a list	You can use indexing to modify or assign new values to specific elements in the list.	Example: 1 1 2 2 3 3 4 4 1 my_list = [10, 20, 30, 40, 50] 2 my_list[1] = 25 # Modifying the second element 3 print(my_list) 4 # Output: [10, 25, 30, 40, 50]

pop()	`pop()` method is another way to remove an element from a list in Python. It removes and returns the element at the specified index. If you don't provide an index to the `pop()` method, it will remove and return the last element of the list by default	Example 1: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 1 my_list = [10, 20, 30, 40, 50] 2 removed_element = my_list.pop(2) # Removes and returns the element at index 2 3 print(removed_element) 4 # Output: 30 5 6 print(my_list) 7 # Output: [10, 20, 40, 50]

Example 2:

1. 1
2. 2

3. 3
4. 4
5. 5
6. 6
7. 7
8. my_list = [10, 20, 30, 40, 50]
9. removed_element = my_list.pop() # Removes and returns the last element
10. print(removed_element)
11. # Output: 50
12.
13. print(my_list)
14. # Output: [10, 20, 30, 40]

remove()	To remove an element from a list. The `remove()` method removes the first occurrence of the specified value.	Example: 1 1 2 2 3 3 4 4 1 my_list = [10, 20, 30, 40, 50] 2 my_list.remove(30) # Removes the element 30 3 print(my_list) 4 # Output: [10, 20, 40, 50]

reverse()	The `reverse()` method is used to reverse the order of elements in a list	Example 1: 1 1 2 2 3 3 1 my_list = [1, 2, 3, 4, 5] 2 my_list.reverse() print(my_list) 3 # Output: [5, 4, 3, 2, 1]
Slicing	You can use slicing to access a range of elements from a list.	Syntax: 1 1 1 list_name[start:end:step]

Example:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. my_list = [1, 2, 3, 4, 5]
14. print(my_list[1:4])
15. # Output: [2, 3, 4] (elements from index 1 to 3)
```

```
16.
17. print(my_list[:3])
18. # Output: [1, 2, 3] (elements from the beginning
    up to index 2)
19.
20. print(my_list[2:])
21. # Output: [3, 4, 5] (elements from index 2 to the
    end)
22.
23. print(my_list[::2])
24. # Output: [1, 3, 5] (every second element)
```

sort()	The `sort()` method is used to sort the elements of a list in ascending order. If you want to sort the list in descending order, you can pass the `reverse=True` argument to the `sort()` method.	Example 1: 1 1 2 2 3 3 4 4 1 my_list = [5, 2, 8, 1, 9] 2 my_list.sort() 3 print(my_list) 4 # Output: [1, 2, 5, 8, 9]

Example 2:

```
1. 1
2. 2
3. 3
4. 4
5. my_list = [5, 2, 8, 1, 9]
```

```
6. my_list.sort(reverse=True)
7. print(my_list)
8. # Output: [9, 8, 5, 2, 1]
```

Dictionary

Package/ Method	Description	Code Example
Accessing Values	You can access the values in a dictionary using their corresponding `keys`.	Syntax: 1 1 1 Value = dict_name["key_name"]

Example:

```
1. 1
2. 2
3. name = person["name"]
4. age = person["age"]
```

Add or modify	Inserts a new key-value pair into the dictionary. If the key already exists, the value will be updated; otherwise, a new entry is created.	Syntax: 1 1 1 dict_name[key] = value

Example:

```
1. 1
2. 2
3. person["Country"] = "USA" # A new entry will be
```

created.
4. person["city"] = "Chicago" # Update the existing value for the same key

clear()	The `clear()` method empties the dictionary, removing all key-value pairs within it. After this operation, the dictionary is still accessible and can be used further.	Syntax: 1 1 1 dict_name.clear()

Example:
1. 1
2. grades.clear()

copy()	Creates a shallow copy of the dictionary. The new dictionary contains the same key-value pairs as the original, but they remain distinct objects in memory.	Syntax: 1 1 1 new_dict = dict_name.copy()

Example:
1. 1

2. 2
3. new_person = person.copy()
4. new_person = dict(person) # another way to create a copy of dictionary

Creating a Dictionary	A dictionary is a built-in data type that represents a collection of key-value pairs. Dictionaries are enclosed in curly braces `{} `.	Example: 1 1 2 2 1 dict_name = {} #Creates an empty dictionary 2 person = { "name": "John", "age": 30, "city": "New York"}
del	Removes the specified key-value pair from the dictionary. Raises a `KeyError` if the key does not exist.	Syntax: 1 1 1 del dict_name[key]

Example:

1. 1
2. del person["Country"]

items()	Retrieves all key-value pairs as tuples and converts them into a list of tuples. Each tuple consists of a key and its corresponding value.	Syntax: 1 1 1 items_list = list(dict_name.items())

Example:

```
1. 1
2. info = list(person.items())
```

key existence	You can check for the existence of a key in a dictionary using the `in` keyword	Example: 1 1 2 2 1 if "name" in person: 2 print("Name exists in the dictionary.")

keys()	Retrieves all keys from the dictionary and converts them into a list. Useful for iterating or processing keys using list methods.	Syntax: 1 1 1 keys_list = list(dict_name.keys())

Example:

```
1. 1
2. person_keys = list(person.keys())
```

update()	The `update()` method merges the provided dictionary into the existing dictionary, adding or updating key-value pairs.	Syntax: 1 1 1 dict_name.update({key: value})

Example:

```
1. 1
2. person.update({"Profession": "Doctor"})
```

values()	Extracts all values from the dictionary and converts them into a list. This list can be used for further processing or analysis.	Syntax: 1 1 1 values_list = list(dict_name.values())

Example:

```
1. 1
2. person_values = list(person.values())
```

Sets

Package/ Method	Description	Code Example
add()	Elements can be added to a set using the `add()` method. Duplicates are automatically removed, as sets only store unique values.	Syntax: 1 1 1 set_name.add(element)

Example:

```
1. 1
2. fruits.add("mango")
```

clear()	The `clear()` method removes all elements from the set, resulting in an empty set. It updates the set in-place.	Syntax: 1 1 1 set_name.clear()

Example:

```
1. 1
2. fruits.clear()
```

copy()	The `copy()` method creates a shallow copy of the set. Any modifications to the copy won't affect the original set.	Syntax: 1 1 1 new_set = set_name.copy()

Example:

```
1. 1
2. new_fruits = fruits.copy()
```

Defining Sets	A set is an unordered collection of unique elements. Sets are enclosed in curly braces `{} `. They are useful for storing distinct values and performing set operations.	Example: 1 1 2 2 1 empty_set = set() #Creating an Empty Set 2 fruits = {"apple", "banana", "orange"}
discard()	Use the `discard()` method to remove a specific element from the set. Ignores if the element is not found.	Syntax: 1 1 1 set_name.discard(element)

Example:

1. 1
2. fruits.discard("apple")

issubset()	The `issubset()` method checks if the current set is a subset of another set. It returns True if all elements of the current set are present in the other set, otherwise False.	Syntax: 1 1 1 is_subset = set1.issubset(set2)

Example:

```
1. 1
2. is_subset = fruits.issubset(colors)
```

issuperset()	The `issuperset()` method checks if the current set is a superset of another set. It returns True if all elements of the other set are present in the current set, otherwise False.	Syntax: 1 1 1 is_superset = set1.issuperset(set2)

Example:

```
1. 1
2. is_superset = colors.issuperset(fruits)
```

pop()	The `pop()` method removes and returns an arbitrary element from the set. It raises a `KeyError` if the set is empty. Use this method to remove elements when the order doesn't matter.	Syntax: 1 1 1 removed_element = set_name.pop()

Example:

```
1. 1
2. removed_fruit = fruits.pop()
```

remove()	Use the `remove()` method to remove a specific element from the set. Raises a `KeyError` if the element is not found.	Syntax: 1 1 1 set_name.remove(element)

Example:

```
1. 1
2. fruits.remove("banana")
```

Set Operations	Perform various operations on sets: `union`, `intersection`, `difference`, `symmetric difference`.	Syntax: 1 1 2 2 3 3 4 4 1 union_set = set1.union(set2) 2 intersection_set = set1.intersection(set2) 3 difference_set = set1.difference(set2) 4 sym_diff_set = set1.symmetric_difference(set2)

Example:

1. 1
2. 2
3. 3
4. 4
5. combined = fruits.union(colors)
6. common = fruits.intersection(colors)
7. unique_to_fruits = fruits.difference(colors)
8. sym_diff = fruits.symmetric_difference(colors)

update()	The `update()` method adds elements from another iterable into the set. It maintains the uniqueness of elements.	Syntax: 1 1 1 set_name.update(iterable)

Example:

```
1. 1
2. fruits.update(["kiwi", "grape"]
```

-

Cheat Sheet: Python Data Structures Part-2

Dictionaries

2023-10-11 14:37:21 Wednesday

Package/ Method	Description	Code Example
Creating a Dictionary	A dictionary is a built-in data type that represents a collection of key-value pairs. Dictionaries are enclosed in curly braces {}.	Example: 1 1 2 2 1 dict_name = {} #Creates an empty dictionary 2 person = { "name": "John", "age": 30, "city": "New York"}
Accessing Values	You can access the values in a dictionary using their corresponding keys.	Syntax: 1 1 1 Value = dict_name["key_name"]

Example:

```
1. 1
2. 2
3. name = person["name"]
4. age = person["age"]
```

Add or modify	Inserts a new key-value pair into the dictionary. If the key already exists, the value will be updated; otherwise, a new entry is created.	Syntax: 1 1 1 dict_name[key] = value

Example:

```
1
2
person["Country"] = "USA" # A new entry will be created.
person["city"] = "Chicago"  # Update the existing value for the same key
```

del	Removes the specified key-value pair from the dictionary. Raises a KeyError if the key does not exist.	Syntax: 1 1 1 del dict_name[key]

Example:

```
1
del person["Country"]
```

update()	The update() method merges the provided dictionary into the existing dictionary, adding or updating key-value pairs.	Syntax: 1 1 1 dict_name.update({key: value})

Example:

```
1. 1
2. person.update({"Profession": "Doctor"})
```

clear()	The clear() method empties the dictionary, removing all key-value pairs within it. After this operation, the dictionary is still accessible and can be used further.	Syntax: 1 1 1 dict_name.clear()

Example:

```
1. 1
2. grades.clear()
```

key existence	You can check for the existence of a key in a dictionary using the in keyword	Example: 1 1 2 2 1 if "name" in person: 2 print("Name exists in the dictionary.")
copy()	Creates a shallow copy of the dictionary. The new dictionary contains the same key-value pairs as the original, but they remain distinct objects in memory.	Syntax: 1 1 1 new_dict = dict_name.copy()

Example:

```
1
2
new_person = person.copy()
new_person = dict(person) # another way to create a copy of dictionary
```

keys()	Retrieves all keys from the dictionary and converts them into a list. Useful for iterating or processing keys using list methods.	Syntax: 1 1 1 keys_list = list(dict_name.keys())

Example:

```
1. 1
2. person_keys = list(person.keys())
```

values()	Extracts all values from the dictionary and converts them into a list. This list can be used for further processing or analysis.	Syntax: 1 1 1 values_list = list(dict_name.values())

Example:

```
1. 1
2. person_values = list(person.values())
```

items()	Retrieves all key-value pairs as tuples and converts them into a list of tuples. Each tuple consists of a key and its corresponding value.	Syntax: 1 1 1 items_list = list(dict_name.items())

Example:

```
1. 1
2. info = list(person.items())
```

Sets

Package/ Method	Description	Code Example
add()	Elements can be added to a set using the `add()` method. Duplicates are automatically removed, as sets only store unique values.	Syntax: 1 1 1 set_name.add(element)

Example:

```
1. 1
2. fruits.add("mango")
```

clear()	The `clear()` method removes all elements from the set, resulting in an empty set. It updates the set in-place.	Syntax: 1 1 1 set_name.clear()

Example:

```
1. 1
2. fruits.clear()</td>
```

copy()	The `copy()` method creates a shallow copy of the set. Any modifications to the copy won't affect the original set.	Syntax: 1 1 1 new_set = set_name.copy()

Example:

```
1. 1
2. new_fruits = fruits.copy()
```

Defining Sets	A set is an unordered collection of unique elements. Sets are enclosed in curly braces `{} `. They are useful for storing distinct values and performing set operations.	Example: 1 1 2 2 1 empty_set = set() #Creating an Empty 2 Set fruits = {"apple", "banana", "orange"}
discard()	Use the `discard()` method to remove a specific element from the set. Ignores if the element is not found.	Syntax: 1 1 1 set_name.discard(element)

Example:

1. 1
2. fruits.discard("apple")

issubset()	The `issubset()` method checks if the current set is a subset of another set. It returns True if all elements of the current set are present in the other set, otherwise False.	Syntax: 1 1 1 is_subset = set1.issubset(set2)

Example:

```
1. 1
2. is_subset = fruits.issubset(colors)
```

issuperset()	The `issuperset()` method checks if the current set is a superset of another set. It returns True if all elements of the other set are present in the current set, otherwise False.	Syntax: is_superset = set1.issuperset(set2) Example: 1 1 1 is_superset = colors.issuperset(fruits)

pop()	The `pop()` method removes and returns an arbitrary element from the set. It raises a `KeyError` if the set is empty. Use this method to remove elements when the order doesn't matter.	Syntax: 1 1 1 removed_element = set_name.pop()

Example:

```
1. 1
2. removed_fruit = fruits.pop()
```

remove()	Use the `remove()` method to remove a specific element from the set. Raises a `KeyError` if the element is not found.	Syntax: 1 1 1 set_name.remove(element)

Example:

```
1. 1
2. fruits.remove("banana")
```

Set Operations	Perform various operations on sets: `union`, `intersection`, `difference`, `symmetric difference`.	Syntax: 1 1 2 2 3 3 4 4 1 union_set = set1.union(set2) 2 intersection_set = set1.intersection(set2) 3 difference_set = set1.difference(set2) 4 sym_diff_set = set1.symmetric_difference(set2)

Example:

1. 1
2. 2
3. 3
4. 4
5. combined = fruits.union(colors)
6. common = fruits.intersection(colors)
7. unique_to_fruits = fruits.difference(colors)
8. sym_diff = fruits.symmetric_difference(colors)

update()	The `update()` method adds elements from another iterable into the set. It maintains the uniqueness of elements.	Syntax: 1 1 1 set_name.update(iterable)

Example:

```
1. 1
2. fruits.update(["kiwi", "grape"])
```


•

% buffered00:00
03:52

Conditions and Branching

Estimated time needed: 10 minutes

Objective:

In this reading, you'll learn about:

1. Comparison operators
2. Branching
3. Logical operators

1. Comparison operations

Comparison operations are essential in programming. They help compare values and make decisions based on the results.

Equality operator

The equality operator == checks if two values are equal. For example, in Python:

```
1
2
3
age = 25
if age == 25:
    print("You are 25 years old.")
```

Here, the code checks if the variable age is equal to 25 and prints a message accordingly.

Inequality operator

The inequality operator != checks if two values are not equal:

```
1
```

```
2
if age != 30:
    print("You are not 30 years old.")
```

Here, the code checks if the variable age is not equal to 30 and prints a message accordingly.

Greater than and less than

You can also compare if one value is greater than another.

```
1
2
if age>= 20:
    Print("Yes, the Age is greater than 20")
```

Here, the code checks if the variable age is greater than or equal to 20 and prints a message accordingly.

2. Branching

Branching is like making decisions in your program based on conditions. Think of it as real-life choices.

The IF statement

Consider a real-life scenario of entering a bar. If you're above a certain age, you can enter; otherwise, you cannot.

```
1
2
3
4
5
age = 20
if age >= 21:
    print("You can enter the bar.")
else:
    print("Sorry, you cannot enter.")
```

Here, you are using the if statement to make a decision based on the age variable.

The ELIF Statement

Sometimes, there are multiple conditions to check. For example, if you're not old enough for the bar, you can go to a movie instead.

```
1
2
3
4
5
6
if age >= 21:
    print("You can enter the bar.")
elif age >= 18:
    print("You can watch a movie.")
else:
    print("Sorry, you cannot do either.")
```

Real-life example: Automated Teller Machine (ATM)

When a user interacts with an ATM, the software in the ATM can use branching to make decisions based on the user's input. For example, if the user selects "Withdraw Cash" the ATM can branch into different denominations of bills to dispense based on the amount requested.

```
1
2
3
4
5
6
7
8
9
user_choice = "Withdraw Cash"
if user_choice == "Withdraw Cash":
```

```
    amount = input("Enter the amount to withdraw: ")
    if amount % 10 == 0:
        dispense_cash(amount)
    else:
        print("Please enter a multiple of 10.")
else:
    print("Thank you for using the ATM.")
```

3. Logical operators

Logical operators help combine and manipulate conditions.

The NOT operator

Real-life example: Notification settings

In a smartphone's notification settings, you can use the NOT operator to control when to send notifications. For example, you might only want to receive notifications when your phone is not in "Do Not Disturb" mode.

The not operator negates a condition.

```
1
2
3
is_do_not_disturb = True
if not is_do_not_disturb:
    send_notification("New message received")
```

The AND operator

Real-life example: Access control

In a secure facility, you can use the AND operator to check multiple conditions for access. To open a high-security door, a person might need both a valid ID card and a matching fingerprint.

The AND operator checks if all required conditions are true, like needing both keys to open a safe.

```
1
```

```
2. 2
3. 3
4. 4
5. has_valid_id_card = True
6. has_matching_fingerprint = True
7. if has_valid_id_card and
   has_matching_fingerprint:
8.     open_high_security_door()
```

The OR operator

Real-life example: Movie night decision

When planning a movie night with friends, you can use the OR operator to decide on a movie genre. You'll choose a movie if at least one person is interested. The OR operator checks if at least one condition is true. It's like choosing between different movies to watch.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. friend1_likes_comedy = True
7. friend2_likes_action = False
8. friend3_likes_drama = False
9. if friend1_likes_comedy or friend2_likes_action
   or friend3_likes_drama:
10.     choose a movie()
```

Summary

In this reading, you delved into the most frequently used operator and the concept of conditional branching, which encompasses the utilization of if statements and if-else statements.

Author

Akansha Yadav

•

Introduction to Loops in Python

Estimated time needed: 10 minutes

Objectives

1. Understand Python loops.
2. How the loop Works
3. Learn about the needs for loop
4. Utilize Python's Range function.
5. Familiarize with Python's enumerate function.
6. Apply while loops for conditional tasks.
7. Distinguish appropriate loop selection.

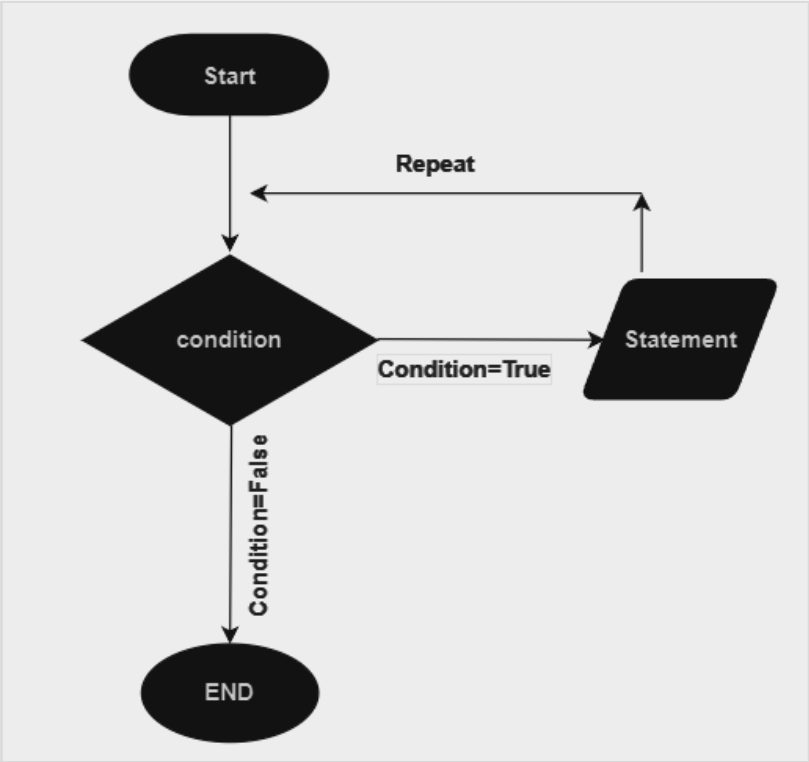
What is a Loop?

In programming, a loop is like a magic trick that allows a computer to do something over and over again. Imagine you are a magician's assistant, and your magician friend asks you to pull a rabbit out of a hat, but not just once - they want you to keep doing it until they tell you to stop. That is what loops do for computers - they repeat a set

of instructions as many times as needed.

How Loop works?

Here's how it works in Python:



- **Start:** The for loop begins with the keyword for, followed by a variable that will take on each value in a sequence.
- **Condition:** After the variable, you specify the keyword in and a sequence, such as a list or a range, that the loop will iterate through.
- If **Condition True**:

1. The loop takes the first value from the sequence and assigns it to the variable.
2. The indented block of code following the loop header is executed using this value.
3. The loop then moves to the next value in the sequence and repeats the process until all values have been used.

- **Statement:** Inside the indented block of the loop, you write the statements that you want to repeat

for each value in the sequence.

- **Repeat:** The loop continues to repeat the block of code for each value in the sequence until there are no more values left.
- If **Condition False**:

1. Once all values in the sequence have been processed, the loop terminates automatically.
2. The loop completes its execution, and the program continues to the next statement after the loop.

The Need for Loops

Think about when you need to count from 1 to 10. Doing it manually is easy, but what if you had to count to a **million**? Typing all those numbers one by one would be a nightmare! This is where loops come in handy. They help computers repeat tasks quickly and accurately without getting tired.

Main Types of Loops

For Loops

For loops are like a superhero's checklist. A for loop in programming is a control structure that allows the repeated execution of a set of statements for each item in a sequence, such as elements in a list or numbers in a range, enabling efficient iteration and automation of tasks

Syntax of for loop

1. 1
2. 2
3. for val in sequence:
4. # statement(s) to be executed in sequence as a part of the loop.

Here is an example of For loop.

Imagine you're a painter, and you want to paint a beautiful rainbow with seven colors. Instead of picking up each color one by one and painting the rainbow, you could tell a magical painter's assistant to do it for you. This is what a basic for loop does in programming.

We have a list of colours.

```
1
colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]
```

Let's print the colour name in the new line using for loop.

```
1
2
for color in colors:
    print(color)
```

In this example, the for loop picks each color from the colors list and prints it on the screen. You don't have to write the same code for each color - the loop does it automatically!

Sometimes you do not want to paint a rainbow, but *you want to count the number of steps to reach your goal*. A range-based for loop is like having a friendly step counter that helps you reach your target.

Here is how you might use a for loop to count from 1 to 10:

```
1
2
for number in range(1, 11):
    print(number)
```

Here, the **range(1, 11)** generates a sequence from 1 to 10, and the for loop goes through each number in that sequence, printing it out. It's like taking 10 steps, and you're guided by the loop!

Range Function

The range function in Python generates an ordered sequence that can be used in loops. It takes one or two arguments:

- If given one argument (e.g., range(11)), it generates a sequence starting from 0 up to (but not including) the given number.

```
1
2
for number in range(11):
    print(number)
```

- If given two arguments (e.g., range(1, 11)), it generates a sequence starting from the first argument up to (but not including) the second argument.

```
1
2
for number in range(1, 11):
    print(number)
```

The Enumerated For Loop

Have you ever needed to keep track of both the item and its position in a list? An enumerated for loop comes to your rescue. It's like having a personal assistant who not only hands you the item but also tells you where to find it.

Consider this example:

```
1
2
3
fruits = ["apple", "banana", "orange"]
for index, fruit in enumerate(fruits):
    print(f"At position {index}, I found a {fruit}")
```

With this loop, you not only get the fruit but also its

position in the list. It's as if you have a magical guide pointing out each fruit's location!

While Loops

While loops are like a sleepless night at a friend's sleepover. Imagine you and your friends keep telling ghost stories until someone decides it's time to sleep. As long as no one says, "Let's sleep" you keep telling stories.

A while loop works similarly - it repeats a task as long as a certain condition is true. It's like saying, "Hey computer, keep doing this until I say stop!"

Basic syntax of While Loop.

```
1
2
3
while condition:
    # Code to be executed while the condition is true
    # Indentation is crucial to indicate the scope of the loop
```

For example, here's how you might use a while loop to count from 1 to 10:

```
1
2
3
4
count = 1
while count <= 10:
    print(count)
    count += 1
```

here's a breakdown of the above code.

1. There is a variable named **count** initialized with the value 1.

2. The while loop is used to repeatedly execute a block of code as long as a given condition is True. In this case, the condition is **count <= 10**, meaning the loop will continue as long as count is less than or equal to 10.
3. Inside the loop:
    - The **print(count)** statement outputs the current value of the count variable.
    - The **count += 1** statement increments the value of count by 1. This step ensures that the loop will eventually terminate when count becomes greater than 10.
4. The loop will continue executing as long as the condition count <= 10 is satisfied.
5. The loop will print the numbers 1 to 10 in consecutive order since the print statement is inside the loop block and executed during each iteration.
6. Once count reaches 11, the condition count <= 10 will evaluate to False, and the loop will terminate.
7. The output of the code will be the numbers 1 to 10, each printed on a separate line.

The Loop Flow

Both for and while loops have their special moves, but they follow a pattern:

- **Initialization:** You set up things like a starting point or conditions.
- **Condition:** You decide when the loop should keep going and when it should stop.
- **Execution:** You do the task inside the loop.
- **Update:** You make changes to your starting point or conditions to move forward.

- **Repeat:** The loop goes back to step 2 until the condition is no longer true.

When to Use Each

For Loops: Use for loops when you know the number of iterations in advance and want to process each element in a sequence. They are best suited for iterating over collections and sequences where the length is known.

While Loops: Use while loops when you need to perform a task repeatedly as long as a certain condition holds true. While loops are particularly useful for situations where the number of iterations is uncertain or where you're waiting for a specific condition to be met.

Summary

In this adventure into coding, we explored loops in Python - special tools that help us do things over and over again without getting tired. We met two types of loops: **"for loops"** and **"while loops."**

For Loops were like helpers that made us repeat tasks in order. We painted colors, counted numbers, and even got a helper to tell us where things were in a list. For loops made our job easier and made our code look cleaner.

While Loops were like detectives that kept doing something as long as a rule was true. They helped us take steps, guess numbers, and work until we were tired. While loops were like smart assistants that didn't stop until we said so.

Author(s)

Akansha Yadav

•

Exploring Python Functions

Estimated time needed: 15 minutes

Objectives:

By the end of this reading, you should be able to:

1. Describe the function concept and the importance of functions in programming
2. Write a function that takes inputs and performs tasks
3. Use built-in functions like len(), sum(), and others effectively
4. Define and use your functions in Python
5. Differentiate between global and local variable scopes
6. Use loops within the function
7. Modify data structures using functions

Introduction to functions

A function is a fundamental building block that encapsulates specific actions or computations. As in mathematics, where functions take inputs and produce outputs, programming functions perform similarly. They take inputs, execute predefined actions or calculations, and then return an output.

Purpose of functions

Functions promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever you need that task. This reduces redundancy and makes the code easier to

manage and maintain.

Benefits of using functions

Modularity: Functions break down complex tasks into manageable components

Reusability: Functions can be used multiple times without rewriting code

Readability: Functions with meaningful names enhance code understanding

Debugging: Isolating functions eases troubleshooting and issue fixing

Abstraction: Functions simplify complex processes behind a user-friendly interface

Collaboration: Team members can work on different functions concurrently

Maintenance: Changes made in a function automatically apply wherever it's used

How functions take inputs, perform tasks, and produce outputs

Inputs (Parameters)

Functions operate on data, and they can receive data as input. These inputs are known as *parameters* or *arguments*. Parameters provide functions with the necessary information they need to perform their tasks. Consider parameters as values you pass to a function, allowing it to work with specific data.

Performing tasks

Once a function receives its input (parameters), it executes predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.

Producing outputs

After performing its tasks, a function can produce an

output. This output is the result of the operations carried out within the function. It's the value that the function “returns” to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.
Example:
Consider a function named calculate_total that takes two numbers as input (parameters), adds them together, and then produces the sum as the output.
Here's how it works:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. def calculate_total(a, b): # Parameters: a and b
8. total = a + b # Task: Addition
9. return total # Output: Sum of a and b
10.
11. result = calculate_total(5, 7) # Calling the function with inputs 5 and 7
12. print(result) # Output: 12

Python's built-in functions
Python has a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.
Using built-in functions or Pre-defined functions
To use a built-in function, you simply call the function's

name followed by parentheses. Any required arguments or parameters are passed into the function within these parentheses. The function then performs its predefined task and may return an output you can use in your code. Here are a few examples of commonly used built-in functions:

len(): Calculates the length of a sequence or collection

```
1
2
string_length = len("Hello, World!")  # Output: 13
list_length = len([1, 2, 3, 4, 5])   # Output: 5
```

sum(): Adds up the elements in an iterable (list, tuple, and so on)

```
1
total = sum([10, 20, 30, 40, 50])  # Output: 150
```

max(): Returns the maximum value in an iterable

```
1
highest = max([5, 12, 8, 23, 16])  # Output: 23
```

min(): Returns the minimum value in an iterable

```
1
lowest = min([5, 12, 8, 23, 16])  # Output: 5
```

Python's built-in functions offer a wide array of functionalities, from basic operations like len() and sum() to more specialized tasks.

Defining your functions

Defining a function is like creating your mini-program:

1. Use def followed by the function name and parentheses

Here is the syntax to define a function:

```
1
2
def function_name():
    pass
```

A "pass" statement in a programming function is a placeholder or a no-op (no operation) statement. Use it when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.

- **Placeholder:** "pass" acts as a temporary placeholder for future code that you intend to write within a function or a code block.
- **Syntax Requirement:** In many programming languages like Python, using "pass" is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.
- **No Operation:** "pass" itself doesn't perform any meaningful action. When the interpreter encounters “pass”, it simply moves on to the next statement without executing any code.

Function Parameters:

- Parameters are like inputs for functions
- They go inside parentheses when defining the function
- Functions can have multiple parameters

Example:

```
1
2
3
4
5
def greet(name):
    print("Hello, " + name)

result = greet("Alice")
```

```
10. print(result)  # Output: Hello, Alice
```

Docstrings (Documentation Strings)

- Docstrings explain what a function does
- Placed inside triple quotes under the function definition
- Helps other developers understand your function

Example:

```
 1. 1
 2. 2
 3. 3
 4. 4
 5. 5
 6. 6
 7. 7
 8. 8
 9. def multiply(a, b):
10.     """
11.     This function multiplies two numbers.
12.     Input: a (number), b (number)
13.     Output: Product of a and b
14.     """
15.     print(a * b)
16. multiply(2,6)
```

Return statement

- Return gives back a value from a function
- Ends the function's execution and sends the result
- A function can return various types of data

Example:

```
1. 1
2. 2
3. 3
```

```
4. 4
5. def add(a, b):
6.     return a + b
7. 
8. sum_result = add(3, 5)  # sum_result gets the
   value 8
```

Understanding scopes and variables

Scope is where a variable can be seen and used:

- **Global Scope:** Variables defined outside functions; accessible everywhere
- **Local Scope:** Variables inside functions; only usable within that function

Example:

Part 1: Global variable declaration

```
1. 1
2. global_variable = "I'm global"
```

This line initializes a global variable called global_variable and assigns it the value "I'm global". Global variables are accessible throughout the entire program, both inside and outside functions.

Part 2: Function definition

```
1. 1
2. 2
3. 3
4. 4
5. def example_function():
6.     local_variable = "I'm local"
7.     print(global_variable)  # Accessing global
   variable
8.     print(local_variable)   # Accessing local variable
```

Here, you define a function called example_function(). Within this function:

- A local variable named local_variable is declared

and initialized with the string value "I'm local." This variable is local to the function and can only be accessed within the function's scope.

- The function then prints the values of both the **global variable (global_variable) and the local variable (local_variable)**. It demonstrates that you can access global and local variables within a function.

Part 3: Function call

```
1
example_function()
```

In this part, you call the example_function() by invoking it. This results in the function's code being executed. As a result of this function call, it will print the values of the global and local variables within the function.

Part 4: Accessing global variable outside the function

```
1
print(global_variable)  # Accessible outside the function
```

After calling the function, you print the value of the global variable global_variable outside the function.

This demonstrates that global variables are accessible inside and outside of functions.

Part 5: Attempting to access local variable outside the function

```
1
# print(local_variable)  # Error, local variable not visible here
```

In this part, you are attempting to print the value of the local variable local_variable outside of the function. However, this line would result in an error.

Local variables are only visible and accessible within the

scope of the function where they are defined. Attempting to access them outside of that scope would raise a "NameError".

Using functions with loops

Functions and loops together

1. Functions can contain code with loops
2. This makes complex tasks more organized
3. The loop code becomes a repeatable function

Example:

```
1
2
3
4
5
def print_numbers(limit):
    for i in range(1, limit+1):
        print(i)

print_numbers(5)  # Output: 1 2 3 4 5
```

Enhancing code organization and reusability

1. Functions group similar actions for easy understanding
2. Looping within functions keeps code clean
3. You can reuse a function to repeat actions

Example

```
1
2
3
4
5
def greet(name):
    return "Hello, " + name
```

```
8.
9. for _ in range(3):
10.     print(greet("Alice"))
```

Modifying data structure using functions

You'll use Python and a list as the data structure for this illustration. In this example, you will create functions to add and remove elements from a list.

Part 1: Initialize an empty list

```
1. 1
2. 2
3. # Define an empty list as the initial data structure
4. my_list = []
```

In this part, you start by creating an empty list named my_list. This empty list serves as the data structure that you will modify throughout the code.

Part 2: Define a function to add elements

```
1. 1
2. 2
3. 3
4. # Function to add an element to the list
5. def add_element(data_structure, element):
6.     data_structure.append(element)
```

Here, you define a function called add_element. This function takes two parameters:

- data_structure: This parameter represents the list to which you want to add an element
- element: This parameter represents the element you want to add to the list

Inside the function, you use the append method to add the provided element to the data_structure, which is assumed to be a list.

Part 3: Define a function to remove elements

```
1. 1
```

```
2
3
4
5
6
# Function to remove an element from the list
def remove_element(data_structure, element):
    if element in data_structure:
        data_structure.remove(element)
    else:
        print(f"{element} not found in the list.")
```

In this part, you define another function called remove_element. It also takes two parameters:

- data_structure: The list from which we want to remove an element
- element: The element we want to remove from the list

Inside the function, you use conditional statements to check if the element is present in the data_structure. If it is, you use the remove method to remove the first occurrence of the element. If it's not found, you print a message indicating that the element was not found in the list.

Part 4: Add elements to the list

```
1
2
3
4
# Add elements to the list using the add_element function
add_element(my_list, 42)
add_element(my_list, 17)
```

8. add_element(my_list, 99)

Here, you use the add_element function to add three elements (42, 17, and 99) to the my_list. These are added one at a time using function calls.

Part 5: Print the current list

1. 1
2. 2
3. # Print the current list
4. print("Current list:", my_list)

This part simply prints the current state of the my_list to the console, allowing us to see the elements that have been added so far.

Part 6: Remove elements from the list

1. 1
2. 2
3. 3
4. # Remove an element from the list using the remove_element function
5. remove_element(my_list, 17)
6. remove_element(my_list, 55) # This will print a message since 55 is not in the list

In this part, you use the remove_element function to remove elements from the my_list. First, you attempt to remove 17 (which is in the list), and then you try to remove 55 (which is not in the list). **The second call to remove_element will print a message indicating that 55 was not found.**

Part 7: Print the updated list

1. 1
2. 2
3. # Print the updated list
4. print("Updated list:", my_list)

Finally, you print the updated my_list to the console.

This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.

Conclusion

Congratulations! You've completed the Reading Instruction Lab on Python functions. You've gained a solid understanding of functions, their significance, and how to create and use them effectively. These skills will empower you to write more organized, modular, and powerful code in your Python projects.

•

% buffered00:00
05:37

Exception Handling in Python

Estimated time needed: 10 Minutes

Objectives

1. Understanding Exceptions
2. Distinguishing Errors from Exceptions
3. Familiarity with Common Python Exceptions
4. Managing Exceptions Effectively

In the world of programming, errors and unexpected situations are certain. Python, a popular and versatile programming language, equips developers with a powerful toolset to manage these unforeseen scenarios through exceptions and error handling.

What are exceptions?

Exceptions are alerts when something unexpected happens while running a program. It could be a mistake in the code or a situation that was not planned for. Python can raise these alerts automatically, but we can also trigger them on purpose using the raise command. The cool part is that we can prevent our program from crashing by handling exceptions.

Errors vs. Exceptions

Hold on, what is the difference between errors and exceptions? Well, errors are usually big problems that come from the computer or the system. They often make the program stop working completely. On the other hand, exceptions are more like issues we can control. They happen because of something we did in our code and can usually be fixed, so the program keeps going.

Here is the difference between Errors and exceptions:-

Aspect	Errors	Exceptions

Origin	Errors are typically caused by the environment, hardware, or operating system.	Exceptions are usually a result of problematic code execution within the program.
Nature	Errors are often severe and can lead to program crashes or abnormal termination.	Exceptions are generally less severe and can be caught and handled to prevent program termination.
Handling	Errors are not usually caught or handled by the program itself.	Exceptions can be caught using try-except blocks and dealt with gracefully, allowing the program to continue execution.

Examples	Examples include “SyntaxError” due to incorrect syntax or “NameError” when a variable is not defined.	Examples include “ZeroDivisionError” when dividing by zero, or “FileNotFoundError” when attempting to open a non-existent file.
Categorization	Errors are not classified into categories.	Exceptions are categorized into various classes, such as “ArithmeticError,” “IOError,” ValueError,” etc., based on their nature.

Common Exceptions in Python

Here are a few examples of exceptions we often run into and can handle using this tool:

- **ZeroDivisionError:** This error arises when an attempt is made to divide a number by zero. Division by zero is undefined in mathematics, causing an arithmetic error. For instance:
  For example:

1. 1
2. 2
3. 3
4. result = 10 / 0

```
print(result)
# Raises ZeroDivisionError
```

- **ValueError:** This error occurs when an inappropriate value is used within the code. An example of this is when trying to convert a non-numeric string to an integer:
  For example:

```
1
2
num = int("abc")
# Raises ValueError
```

- **FileNotFoundError:** This exception is encountered when an attempt is made to access a file that does not exist.
  For example:

```
1
2
with open("nonexistent_file.txt", "r") as file:
        content = file.read()   # Raises FileNotFoundError
```

- **IndexError:** An IndexError occurs when an index is used to access an element in a list that is outside the valid index range.
  For example:

```
1
2
3
my_list = [1, 2, 3]
value = my_list[1]  # No IndexError, within range
missing = my_list[5]  # Raises IndexError
```

- **KeyError:** The KeyError arises when an attempt is made to access a non-existent key in a dictionary.

For example:

```
1
2
3
my_dict = {"name": "Alice", "age": 30}
value = my_dict.get("city")  # No KeyError, using .get() method
missing = my_dict["city"]  # Raises KeyError
```

- **TypeError:** The TypeError occurs when an object is used in an incompatible manner. An example includes trying to concatenate a string and an integer:
  For example:

```
1
2
result = "hello" + 5
# Raises TypeError
```

- **AttributeError:** An AttributeError occurs when an attribute or method is accessed on an object that doesn't possess that specific attribute or method. For instance:
  For example:

```
1
2
3
text = "example"
length = len(text)  # No AttributeError, correct method usage
missing = text.some_method()  # Raises AttributeError
```

- **ImportError:** This error is encountered when an attempt is made to import a module that is

unavailable. For example: import non_existent_module

Note: **Please remember, the exceptions you will encounter are not limited to just these. There are many more in Python. However, there is no need to worry. By using the technique provided below and following the correct syntax, you will be able to handle any exceptions that come your way.**

Handling Exceptions:
Python has a handy tool called try and except that helps us manage exceptions.
Try and Except : You can use the try and except blocks to prevent your program from crashing due to exceptions.
Here's how they work:

1. The code that may result in an exception is contained in the try block.
2. If an exception occurs, the code directly jumps to except block.
3. In the except block, you can define how to handle the exception gracefully, like displaying an error message or taking alternative actions.
4. After the except block, the program continues executing the remaining code.

Example: Attempting to divide by zero

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6

```
7
8
9
# using Try- except
try:
    # Attempting to divide 10 by 0
    result = 10 / 0
except ZeroDivisionError:
    # Handling the ZeroDivisionError and printing an error message
    print("Error: Cannot divide by zero")
# This line will be executed regardless of whether an exception occurred
print("outside of try and except block")
```

Next Step

As we finish up this reading, you are ready to move on to the next part where you will practice handling errors. For better learning, try out different types of data in the lab. This way, you will encounter various errors and learn how to deal with them effectively. This knowledge will help you write stronger and more reliable code in the future.

Author(s)

Akansha Yadav

% buffered00:00
10:27

Python Objects and Classes

Estimated time needed: 10 minutes

Objectives

In this reading, you will learn about:

- Fundamental concepts of Python objects and classes.
- Structure of classes and object code.
- Real-world examples related to objects and classes.

Introduction to classes and object

Python is an object-oriented programming (OOP) language that uses a paradigm centered around objects and classes.

Let's look at these fundamental concepts.

Classes

A class is a blueprint or template for creating objects. It defines the structure and behavior that its objects will

have.
Think of a class as a cookie cutter and objects as the cookies cut from that template.
In Python, you can create classes using the class keyword.

Creating classes

When you create a class, you specify the attributes(data) and methods (functions) that objects of that class will have.
Attributes are defined as variables within the class, and methods are defined as functions.
For example,you can design a "Car" class with attributes such as "color" and "speed," along with methods like "accelerate."

Objects

An *object* is a fundamental unit in Python that represents a real-world entity or concept.
Objects can be tangible (like a car) or abstract (like a student's grade).
Every object has two main characteristics:

State

The *attributes or data* that describe the object. For your "Car" object, this might include attributes like "color", "speed", and "fuel level".

Behavior

The *actions or methods* that the object can perform. In Python, methods are functions that belong to objects and can change the object's state or perform specific operations.

Instantiating objects

- Once you've defined a class, you can create individual objects (instances) based on that class.
- Each object is independent and has its own set of attributes and methods.

- To create an object, you use the class name followed by parentheses, so: "my_car = Car()"

Interacting with objects

You interact with objects by calling their methods or accessing their attributes using dot notation.

For example, if you have a Car object named **my_car**, you can set its color with **my_car.color = "blue"** and accelerate it with **my_car.accelerate()** if there's an accelerate method defined in the class.

Structure of classes and object code

Please don't directly copy and use this code because it is a template for explanation and not for specific results.

Class declaration (class ClassName)

- The class keyword is used to declare a class in Python.
- ClassName is the name of the class, typically following CamelCase naming conventions.

1. 1
2. class ClassName:

Class attributes (class_attribute = value)

- Class attributes are variables shared among all class instances (objects).
- They are defined within the class but outside of any methods.

1. 1
2. 2
3. 3
4. class ClassName:
5. # Class attributes (shared by all instances)
6. class_attribute = value

Constructor method (def init(self, attribute1, attribute2, ...):)

- The __init__ method is a special method known

as the constructor.

- It initializes the **instance attributes** (also called instance variables) when an object is created.
- The self parameter is the first parameter of the constructor, referring to the instance being created.
- **attribute1**, **attribute2**, and so on are parameters passed to the constructor when creating an object.
- Inside the constructor, self.attribute1, self.attribute2, and so on are used to assign values to instance attributes.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. class ClassName:
10. # Class attributes (shared by all instances)
11. class_attribute = value
12.
13. # Constructor method (initialize instance attributes)
14. def __init__(self, attribute1, attribute2, ...):
15. pass
16. # ...

Instance attributes (self.attribute1 = attribute1)

- Instance attributes are variables that store data specific to each class instance.

- They are initialized within the __init__ method using the self keyword followed by the attribute name.
- These attributes hold unique data for each object created from the class.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. class ClassName:
11. # Class attributes (shared by all instances)
12. class_attribute = value
13.
14. # Constructor method (initialize instance attributes)
15. def __init__(self, attribute1, attribute2, ...):
16. self.attribute1 = attribute1
17. self.attribute2 = attribute2
18. # ...

Instance methods (def method1(self, parameter1, parameter2, ...):)

- Instance methods are functions defined within the class.
- They operate on the instance's data (instance attributes) and can perform actions specific to instances.
- The **self** parameter is required in instance

methods, allowing them to access instance attributes and call other methods within the class.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
        # Method logic
        pass
```

Using the same steps you can define multiple instance

methods.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
```

```
        # Method logic
        pass

    def method2(self, parameter1, parameter2, ...):
        # Method logic
        pass
```

Note: Now, you have successfully created a dummy class.

Creating objects (Instances)

- To create objects (instances) of the class, you call the class like a function and provide arguments the constructor requires.
- Each object is a distinct instance of the class, with its own instance attributes and the ability to call methods defined in the class.

```
1
2
3
# Create objects (instances) of the class
object1 = ClassName(arg1, arg2, ...)
object2 = ClassName(arg1, arg2, ...)
```

Calling methods on objects

- In this section, you will call methods on objects, specifically object1 and object2.
- The methods **method1** and **method2** are defined in the ClassName **class**, and you're calling them on **object1** and **object2** respectively.
- You pass values **param1_value** and **param2_value** as arguments to these methods. These arguments are used within the method's logic.

Method 1: Using dot notation

- This is the most straightforward way to call an object's method. In this, use the dot notation **(object.method())** to invoke the method on the object directly.
- For example, result1 = object1.method1(param1_value, param2_value, ...) calls method1 on object1.

1. 1
2. 2
3. 3
4. 4
5. # Calling methods on objects
6. # Method 1: Using dot notation
7. result1 = object1.method1(param1_value, param2_value, ...)
8. result2 = object2.method2(param1_value, param2_value, ...)

Method 2: Assigning object methods to variables

- Here's an alternative way to call an object's method by assigning the method reference to a variable.
- method_reference = object1.method1 assigns the method **method1** of **object1** to the variable **method_reference**.
- Later, call the method using the variable like this: **result3 = method_reference(param1_value, param2_value, ...)**.

1. 1
2. 2
3. 3
4. # Method 2: Assigning object methods to variables

5. method_reference = object1.method1 # Assign the method to a variable
6. result3 = method_reference(param1_value, param2_value, ...)

Accessing object attributes

- Here, you are accessing an object's attribute using dot notation.
- attribute_value = object1.attribute1 retrieves the value of the attribute **attribute1** from **object1** and assigns it to the variable **attribute_value**.

1. 1
2. 2
3. # Accessing object attributes
4. attribute_value = object1.attribute1 # Access the attribute using dot notation

Modifying object attributes

- You will modify an object's attribute using dot notation.
- object1.attribute2 = new_value sets the attribute **attribute2** of **object1** to the new value **new_value**.

1. 1
2. 2
3. # Modifying object attributes
4. object1.attribute2 = new_value # Change the value of an attribute using dot notation

Accessing class attributes (shared by all instances)

- Finally, access a class attribute shared by all class instances.
- class_attr_value = ClassName.class_attribute accesses the class attribute class_attribute from the ClassName class and assigns its value to the

variable.
class_attr_value.

```
1. 1
2. 2
3. # Accessing class attributes (shared by all
   instances)
4. class_attr_value = ClassName.class_attribute
```

Real-world example

Let's write a python program that simulates a simple car class, allowing you to create car instances, accelerate them, and display their current speeds.

1. Let's start by defining a Car class that includes the following attributes and methods:

- Class attribute max_speed, which is set to **120 km/h**.
- Constructor method __init__ that takes parameters for the **car's make, model, color, and an optional speed (defaulting to 0)**. This method initializes instance attributes for make, model, color, and speed.
- Method accelerate(self, acceleration) that allows the car to accelerate. If the acceleration does not exceed the max_speed, update the **car's speed** attribute. Otherwise, set the speed to the **max_speed**.
- Method get_speed(self) that returns the current speed of the car.

```
1. 1
2. 2
3. 3
4. 4
5. 5
```

```
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. class Car:
23.     # Class attribute (shared by all instances)
24.     max_speed = 120  # Maximum speed in km/h
25.
26.     # Constructor method (initialize instance
    attributes)
27.     def __init__(self, make, model, color,
    speed=0):
28.         self.make = make
29.         self.model = model
30.         self.color = color
31.         self.speed = speed  # Initial speed is set to 0
32.
33.     # Method for accelerating the car
34.     def accelerate(self, acceleration):
35.         if self.speed + acceleration <=
```

Car.max_speed:
36. self.speed += acceleration
37. else:
38. self.speed = Car.max_speed
39.
40. # Method to get the current speed of the car
41. def get_speed(self):
42. return self.speed
43. Now, you will instantiate two objects of the Car class, each with the following characteristics:

- car1: **Make = "Toyota", Model = "Camry", Color = "Blue"**
- car2: **Make = "Honda", Model = "Civic", Color = "Red"**

1. 1
2. 2
3. 3
4. # Create objects (instances) of the Car class
5. car1 = Car("Toyota", "Camry", "Blue")
6. car2 = Car("Honda", "Civic", "Red")
7. Using the accelerate method, you will increase the speed of car1 by 30 km/h and car2 by 20 km/h.
8. 1
9. 2
10. 3
11. # Accelerate the cars
12. car1.accelerate(30)
13. car2.accelerate(20)
14. Lastly, you will display the current speed of each car by utilizing the get_speed method.
15. 1

```
2
3
# Print the current speeds of the cars
print(f"{car1.make} {car1.model} is currently at {car1.get_speed()} km/h.")
print(f"{car2.make} {car2.model} is currently at {car2.get_speed()} km/h.")
```

Next steps

In conclusion, this reading provides a fundamental understanding of objects and classes in Python, essential concepts in object-oriented programming. Classes serve as blueprints for creating objects, encapsulating data attributes and methods. Objects represent real-world entities and possess their unique state and behavior. The structured code example presented in the reading outlines the key elements of a class, including class attributes, the constructor method for initializing instance attributes, and instance methods for defining object-specific functionality.

In the upcoming laboratory session, you can apply the concepts of objects and classes to gain hands-on experience.

Author

Akansha Yadav

Python Programming Fundamentals Cheat Sheet

Package/ Method	Description	Syntax and Code Example
AND	Returns `True` if both statement1 and statement2 are `True`. Otherwise, returns `False`.	Syntax: 1 1 1 statement1 and statement2

Example:

```
1
2
3
4
5
6
7
8
9
marks = 90
attendance_percentage = 87

if marks >= 80 and attendance_percentage >= 85:
    print("qualify for honors")
else:
    print("Not qualified for honors")

# Output = qualify for honors
```

Class Definition	Defines a blueprint for creating objects and defining their attributes and behaviors.	Syntax: 1 1 1 class ClassName: # Class attributes and methods

Example:

```
1
2
3
4
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Define Function	A `function` is a reusable block of code that performs a specific task or set of tasks when called.	Syntax: 1 1 1 def function_name(parameters): # Function body

Example:

```
1
def greet(name): print("Hello,", name)
```

Equal(==)	Checks if two values are equal.	Syntax: 1 1 1 variable1 == variable2

Example 1:

```
1
5 == 5
```

returns True
Example 2:

```
1
age = 25 age == 30
```

returns False		
For Loop	A `for` loop repeatedly executes a block of code for a specified number of iterations or over a sequence of elements (list, range, string, etc.).	Syntax: 1 1 1 for variable in sequence: # Code to repeat

Example 1:

```
1
2
for num in range(1, 10):
    print(num)
```

Example 2:

```
1
2
3
fruits = ["apple", "banana", "orange", "grape", "kiwi"]
for fruit in fruits:
    print(fruit)
```

Function Call	A function call is the act of executing the code within the function using the provided arguments.	Syntax: 1 1 1 function_name(arguments)

Example:

```
1
greet("Alice")
```

Greater Than or Equal To(>=)	Checks if the value of variable1 is greater than or equal to variable2.	Syntax: 1 1 1 variable1 >= variable2

Example 1:

```
1
5 >= 5 and 9 >= 5
```

returns True

Example 2:

```
1
2
3
quantity = 105
minimum = 100
quantity >= minimum
```

returns True		

Greater Than(>)	Checks if the value of variable1 is greater than variable2.	Syntax: 1 1 1 variable1 > variable2

Example 1: 9 > 6
returns True
Example 2:

```
1
2
3
4 age = 20
5 max_age = 25
6 age > max_age
```

returns False		
If Statement	Executes code block `if` the condition is `True`.	Syntax: 1 1 1 if condition: #code block for if statement

Example:

```
1
2
3 if temperature > 30:
4 print("It's a hot day!")
```

If-Elif-Else	Executes the first code block if condition1 is `True`, otherwise checks condition2, and so on. If no condition is `True`, the else block is executed.	Syntax: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 1 if condition1: 2 # Code if condition1 is True 3 4 elif condition2: 5 # Code if condition2 is True 6 7 else: 8 # Code if no condition is True

Example:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6

```
7
8
9
score = 85  # Example score
if score >= 90:
    print("You got an A!")
elif score >= 80:
    print("You got a B.")
else:
    print("You need to work harder.")

# Output = You got a B.
```

If-Else Statement	Executes the first code block if the condition is `True`, otherwise the second block.	Syntax: 1 1 2 2 1 if condition: # Code, if condition is True 2 else: # Code, if condition is False

Example:

```
1
2
3
4
if age >= 18:
    print("You're an adult.")
else:
```

```
8.     print("You're not an adult yet.")
```

Less Than or Equal To(<=)	Checks if the value of variable1 is less than or equal to variable2.	Syntax: 1 1 1 variable1 <= variable2

Example 1:

```
1. 1
2. 5 <= 5 and 3 <= 5
```

returns True

Example 2:

```
1. 1
2. 2
3. 3
4. size = 38
5. max_size = 40
6. size <= max_size
```

returns True		
Less Than(<)	Checks if the value of variable1 is less than variable2.	Syntax: 1 1 1 variable1 < variable2

Example 1:

```
1. 1
2. 4 < 6
```

returns True

Example 2:

```
1. 1
2. 2
3. 3
4. score = 60
5. passing_score = 65
```

6. score < passing_score

returns True		
Loop Controls	`break` exits the loop prematurely. `continue` skips the rest of the current iteration and moves to the next iteration.	Syntax: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 1 for: # Code to repeat 2 if # boolean statement 3 break 4 5 for: # Code to repeat 6 if # boolean statement 7 continue

Example 1:

1. 1
2. 2
3. 3
4. 4
5. for num in range(1, 6):
6. if num == 3:

```
7.         break
8.     print(num)
```

Example 2:

```
1. 1
2. 2
3. 3
4. 4
5. for num in range(1, 6):
6.     if num == 3:
7.         continue
8.     print(num)
```

NOT	Returns `True` if variable is `False`, and vice versa.	Syntax: 1 1 1 !variable

Example:

```
1. 1
2. !isLocked
```

returns True if the variable is False (i.e., unlocked).		
Not Equal(!=)	Checks if two values are not equal.	Syntax: 1 1 1 variable1 != variable2

Example:

```
1. 1
2. 2
3. 3
4. a = 10
```

5. b = 20
6. a != b

returns True

Example 2:

1. 1
2. 2
3. count=0
4. count != 0

returns False		
Object Creation	Creates an instance of a class (object) using the class constructor.	Syntax: 1 1 1 object_name = ClassName(arguments)

Example:

1. 1
2. person1 = Person("Alice", 25)

OR	Returns `True` if either statement1 or statement2 (or both) are `True`. Otherwise, returns `False`.	Syntax: 1 1 1 statement1 || statement2

Example:

1. 1
2. 2
3. "Farewell Party Invitation"
4. Grade = 12 grade == 11 or grade == 12

returns True		
range()	Generates a sequence of numbers within a specified range.	Syntax: 1 1 2 2 3 3 1 range(stop) 2 range(start, stop) 3 range(start, stop, step)

Example:

```
1
2
3
range(5) #generates a sequence of integers from 0 to 4.
range(2, 10) #generates a sequence of integers from 2 to 9.
range(1, 11, 2) #generates odd integers from 1 to 9.
```

Return Statement	`Return` is a keyword used to send a value back from a function to its caller.	Syntax: 1 1 1 return value

Example:

```
1
```

```
2. 2
3. def add(a, b): return a + b
4. result = add(3, 5)
```

Try-Except Block	Tries to execute the code in the try block. If an exception of the specified type occurs, the code in the except block is executed.	Syntax: 1 1 2 2 1 try: # Code that might raise an exception except 2 ExceptionType: # Code to handle the exception

Example:

```
1. 1
2. 2
3. 3
4. 4
5. try:
6.     num = int(input("Enter a number: "))
7. except ValueError:
8.     print("Invalid input. Please enter a valid
   number.")
```

Try-Except with Else Block	Code in the `else` block is executed if no exception occurs in the try block.	Syntax: 1 1 2 2 3 3 1 try: # Code that might raise an exception except 2 ExceptionType: # Code to handle the exception 3 else: # Code to execute if no exception occurs

Example:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. try:
8.     num = int(input("Enter a number: "))
9. except ValueError:
10.     print("Invalid input. Please enter a valid
    number")
11. else:
```

```
12.     print("You entered:", num)
```

Try-Except with Finally Block	Code in the `finally` block always executes, regardless of whether an exception occurred.	Syntax: 1 1 2 2 3 3 1 try: # Code that might raise an exception except 2 ExceptionType: # Code to handle the exception 3 finally: # Code that always executes

Example:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. try:
9.     file = open("data.txt", "r")
10.    data = file.read()
11. except FileNotFoundError:
12.    print("File not found.")
```

13. finally:
14. file.close()

While Loop	A `while` loop repeatedly executes a block of code as long as a specified condition remains `True`.	Syntax: 1 1 1 while condition: # Code to repeat

Example:

1. 1
2. 2
3. count = 0 while count < 5:
4. print(count) count += 1

-

% buffered00:00
08:09

Reading a file with Open()

Estimated time needed: 10 minutes

File handling is an essential aspect of programming, and Python provides built-in functions to interact with files. This guide will explore how to use Python's open function to read the text files ('.txt' files).

Objectives

1. Describe how to use the open() and read() Python functions to open and read the contents of a text file
2. Explain how to use the with statement in Python
3. Describe how to use the readline() function in Python
4. Explain how to use the seek() function to read specific character(s) in a text file

Introduction

Reading text files involves extracting and processing the data stored within them. Text files can have various structures, and how you read them depends on their format. Here's a general guide on reading text files with different structures.

Plain text files

- Plain text files contain unformatted text without any specific structure
- You can read plain text files line by line or load all

the content into your memory

Opening the file

There are two methods for opening the file using the file handling concept.

1. Using Python's open function

Suppose we have a file named **'file.txt'**.

Python's open function creates a file object and accesses the data within a text file. It takes two primary parameters:

1. **File path**: The file path parameter consists of the filename and directory where the file is located.
2. **Mode**: The mode parameter specifies the purpose of opening the file, such as 'r' for reading, 'w' for writing, or 'a' for appending.
3. 1
4. 2
5. # Open the file in read ('r') mode
6. file = open('file.txt', 'r')

open('file.txt', 'r'):

This line opens a file named 'file.txt' in read mode ('r'). It returns a file object, which is stored in the variable file. The 'r' mode indicates that the file will be opened for reading.

2. Using 'with' statement

To simplify file handling and ensure proper closure of files, Python provides the "with" statement. It automatically closes the file when operations within the indented block are completed. This is considered best practice when working with files.

1. 1
2. 2
3. 3
4. # Open the file using 'with' in read ('r') mode

```
with open('file.txt', 'r') as file:
    # further code
```

Open the file using 'with' in read ('r') mode

with open('file.txt', 'r') as file:

This line opens a file named 'file.txt' in read mode ('r') using the with statement, which is a context manager. The file is automatically closed when the code block inside the with statement exits.

Advantages of using the with statement

The key advantages of using the 'with' statement are:

- **Automatic resource management:** The file is guaranteed to be closed when you exit the with block, even if an exception occurs during processing.
- **Cleaner and more concise code:** You don't need to explicitly call **close()**, making your code more readable and less error-prone.

Note: For most file reading and writing operations in Python, the 'with' statement is recommended.

Let's perform a read operation on a file

1. Reading the entire content

You can read the entire content of a file using the read method, which stores the data as a string in a variable. This content can be printed or further manipulated as needed.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8

```
9
10
11
12
13
14
15
16
17
18
19
# Reading and Storing the Entire Content of a File

# Using the read method, you can retrieve the complete content of a file
# and store it as a string in a variable for further processing or display.

# Step 1: Open the file you want to read
with open('file.txt', 'r') as file:

    # Step 2: Use the read method to read the entire content of the file
    file_stuff = file.read()

    # Step 3: Now that the file content is stored in the variable 'file_stuff',
    # you can manipulate or display it as needed.

    # For example, let's print the content to the console:
```

```
35.     print(file_stuff)
36.
37. # Step 4: The 'with' statement automatically
        closes the file when it's done,
38. # ensuring proper resource management and
        preventing resource leaks.
```

Step 1: Involves opening the file, specifying 'file.txt' as the file to be opened for reading ('r') mode using the with context manager.

Step 2: Utilizes the read() statement on the file object (file) to read the entire file. This content is then stored in the file_stuff variable.

Step 3: Explain that with the content now stored in file_stuff, you can perform various operations on it. In the example provided, the code prints the content to the console, but you can manipulate, analyze, search, or process the text data in file_stuff based on your specific needs.

Step 4: Emphasizes that the with block automatically closes the file when done, ensuring proper resource management and preventing resource leaks. This is a crucial aspect of using the with statement when working with files.

2. Reading the content line by line

Python provides methods to read files line by line:

- The 'readlines' method reads the file line by line and stores each line as an element in a list. The order of lines in the list corresponds to their order in the file.
- The 'readline' method reads individual lines from the file. It can be called multiple times to read subsequent lines.

In Python, the readline() method is like reading a book

one line at a time. Imagine you have a big book and want to read it page by page. readline() helps you do just that with lines of text instead of pages.
Here's how it works:
Opening a file: First, you need to open the file you want to read using the open() function.

```
1
file = open('file.txt', 'r')
```

Reading line by line: Now, you can use readline() to read one line from the file at a time. It's like turning the pages of the book, but here, you're getting one sentence (or line) at each turn.

```
1
2
line1 = file.readline()  # Reads the first line
line2 = file.readline()  # Reads the second line
```

Using the lines: You can do things with each line you read. For example, you can print it, check if it contains specific words, or save it somewhere else.

```
1
2
3
print(line1)  # Print the first line
if 'important' in line2:
    print('This line is important!')
```

Looping through lines: Typically, you use a loop to read lines until no more lines are left. t's like reading the entire book, line by line.

```
1
2
3
4
5
```

```
while True:
    line = file.readline()
    if not line:
        break  # Stop when there are no more lines to read
    print(line)
```

Closing the book: When you're done reading, it's essential to close the file using file.close() to make sure you're not wasting resources.

```
1
file.close()
```

So, In simple terms, **readline()** helps you read a text file line by line, allowing you to work with each line of text as you go. It's like taking one sentence at a time from a book and doing something with it before moving on to the next sentence. Don't forget to close the book when you're done!

3. Reading specific characters

You can specify the number of characters to read using the readlines method. For example, reading the first four characters, the next five, and so on.

Reading specific characters from a text file in Python involves opening the file, navigating to the desired position, and then reading the characters you need. Here's a detailed explanation of how to read specific characters from a file:

Open the File

First, you need to open the file you want to read. Use the open() function with the appropriate file path and mode. For reading, use 'r' mode.

```
1
file = open('file.txt', 'r')
```

Navigate to the intended position (Optional)

If you want to read characters from a specific position in

the file, you can use the seek() method. This method moves the file pointer (like a cursor) to a particular position. The position is specified in bytes, so you'll need to know the byte offset of the characters you want to read.

```
1
file.seek(10)  # Move to the 11th byte (0-based index)
```

Read specific characters

To read specific characters, you can use the read() method with an argument that specifies the number of characters to read. It reads characters starting from the current position of the file pointer.

```
1
characters = file.read(5)  # Read the next 5 characters
```

In this example, it reads the next 5 characters from the current position of the file pointer.

Use the read characters

You can now use the characters variable to work with the specific characters you've read. You can print them, save them, manipulate them, or perform any other actions.

```
1
print(characters)
```

Close the file

It's essential to close the file when you're done to free up system resources and ensure proper file handling.

```
1
file.close()
```

Conclusion

In conclusion, this reading has provided a comprehensive overview of file handling in Python, with a focus on reading text files. File handling is a

fundamental aspect of programming, and Python offers powerful built-in functions and methods to interact with files seamlessly.

Author(s)
Akansha Yadav

-

Writing on a file with Open()

Estimated time needed: 10 minutes

Objective

1. Create and write data to a file in Python
2. Write multiple lines of text to a file using lists and loops
3. Add new information to an already existing file without erasing its content
4. Compare and contrast the different file modes in Python, what they mean, and when to use them

Writing to a file

You can create a new text file and write data to it using Python's open() function. The open() function takes two

main arguments: the file path (including the file name) and the mode parameter, which specifies the operation you want to perform on the file. For writing, you should use the mode 'w' Here's an example:

1. 1
2. 2
3. 3
4. 4
5. 5
6. # Create a new file Example2.txt for writing
7. with open('Example2.txt', 'w') as file1:
8. file1.write("This is line A\n")
9. file1.write("This is line B\n")
10. # file1 is automatically closed when the 'with' block exits

Line 2 explanation: with open('Example2.txt', 'w') as file1:**

- We start by using the open function to open or create a file named Example2.txt for writing ('w' mode).
- The 'w' mode specifies that we intend to write data to the file.
- We use the with statement to ensure that the file is automatically closed when the code block exits. This helps manage resources efficiently.

Line 3 explanation: file1.write("This is line A\n")

- Here, we use the write() method on the file object, file1, to add the text This is line A to the file.
- The \n at the end represents a newline character, which starts a new line in the file.

Line 4 explanation file1.write("This is line" B\n")

- Similarly, we use the write() method again to add the text This is line B to the file on a new line.

Writing multiple lines to a file using a list and loop

In Python, you can use a list to store multiple lines of text and then write these lines to a file using a loop. Here's an example code snippet that demonstrates this:

```
1
2
3
4
5
6
7
8
# List of lines to write to the file
Lines = ["This is line 1", "This is line 2", "This is line 3"]

# Create a new file Example3.txt for writing
with open('Example3.txt', 'w') as file2:
    for line in Lines:
        file2.write(line + "\n")
    # file2 is automatically closed when the 'with' block exits
```

Here's an explanation of the code:

- Line 2: We start by defining a list called Lines, which contains multiple lines of text that we want to write to the file. Each line is a string.
- Line 5: Next, we use the open() function to create a new text file named Example3.txt for writing, 'w' mode. The 'w' mode indicates that we intend to write data to the file.

- Line 6: We then enter a for loop to iterate through each element (line) in the Lines list.
- Line 7: Inside the loop, we use the write() method on the file object file2 to write the current line of text (line) to the file. We add \n at the end of each line to ensure that each line is followed by a newline character, which separates them in the file.
- Line 8: Finally, we add a comment indicating that the file file2 will be automatically closed when the code block within the with statement exits. Properly closing the file is essential for good resource management.

Appending data to an existing file

In Python, you can use the 'a' mode when opening a file to append new data to an existing file without overwriting its contents. Here's an example code snippet that demonstrates this:

```
1
2
3
4
5
6
7
# Data to append to the existing file
new_data = "This is line C"

# Open an existing file Example2.txt for appending
with open('Example2.txt', 'a') as file1:
    file1.write(new_data + "\n")
```

```
14.     # file1 is automatically closed when the 'with'
        block exits
```

Here's an explanation of the code:

- Line 2: We start by defining a variable new_data that contains the text we want to append to the existing file. In this case, it's the string `This is line C.``
- Line 5: Next, we use the open() function to open an existing file named Example2.txt for appending, 'a' mode. The 'a' mode indicates that we intend to append data to the file, and if the file doesn't exist, it will be created.
- Line 6: Within the with block, we use the write() method on the file object file1 to append the new_data to the file. We add "\n" at the end to ensure that the appended data starts on a new line, maintaining the file's readability.
- Finally, we add a comment indicating that the file file1 will automatically close when the code block within the with statement exits. Properly closing the file is essential for good resource management.

Copying contents from one file to another

In Python, you can copy the contents of one file to another by reading from the source file and writing to the destination file. Here's an example code snippet that demonstrates this:

1. 1
2. 2
3. 3
4. 4
5. 5

```
6. 6
7. 7
8. 8
9. 9
10. # Open the source file for reading
11. with open('source.txt', 'r') as source_file:
12.     # Open the destination file for writing
13.     with open('destination.txt', 'w') as
    destination_file:
14.         # Read lines from the source file and copy
    them to the destination file
15.         for line in source_file:
16.             destination_file.write(line)
17.     # Destination file is automatically closed when
    the 'with' block exits
18. # Source file is automatically closed when the
    'with' block exits
```

Here's an explanation of the code:

- Line 2: We start by opening the source file, source.txt for reading, r mode, using the with statement and the open() function. This allows us to read data from the source file.
- Line 4: Inside the first with block, we open the destination file, destination.txt for writing, w mode, using another with statement and the open() function. This prepares the destination file for writing.
- Line 6: We use a for loop to iterate through each line in the source file source_file. This loop reads each line from the source file one by one.
- Line 7: Within the loop, we use the write() method to write each line from the source file to the

destination file destination_file. This effectively copies the content of the source file to the destination file.

- Lines 8 and 9: After copying all the lines, both the source and destination files are automatically closed when their respective with blocks exit. Proper file closure is crucial for managing resources efficiently.

File modes in Python (syntax and use cases)

The following table provides an overview of different file modes, their syntax, and common use cases. Understanding these modes is essential when working with files in Python for various data manipulation tasks.

Mode	Syntax	Description
‘r’	'r'	Read mode. Opens an existing file for reading. Raises an error if the file doesn't exist.
‘w’	'w'	Write mode. Creates a new file for writing. Overwrites the file if it already exists.

‘a’	'a'	Append mode. Opens a file for appending data. Creates the file if it doesn't exist.
‘x’	'x'	Exclusive creation mode. Creates a new file for writing but raises an error if the file already exists.
‘rb’	'rb'	Read binary mode. Opens an existing binary file for reading.
‘wb’	'wb'	Write binary mode. Creates a new binary file for writing.
‘ab’	'ab'	Append binary mode. Opens a binary file for appending data.

‘xb’	'xb'	Exclusive binary creation mode. Creates a new binary file for writing but raises an error if it already exists.
‘rt’	'rt'	Read text mode. Opens an existing text file for reading. (Default for text files)
‘wt’	'wt'	Write text mode. Creates a new text file for writing. (Default for text files)
‘at’	'at'	Append text mode. Opens a text file for appending data. (Default for text files)

‘xt’	'xt'	Exclusive text creation mode. Creates a new text file for writing but raises an error if it already exists.
‘r+’	'r+'	Read and write mode. Opens an existing file for both reading and writing.
‘w+’	'w+'	Write and read mode. Creates a new file for reading and writing. Overwrites the file if it already exists.
‘a+’	'a+'	Append and read mode. Opens a file for both appending and reading. Creates the file if it doesn't exist.

‘x+’	'x+'	Exclusive creation and read/write mode. Creates a new file for reading and writing but raises an error if it already exists.

Conclusion

Working with files is a fundamental aspect of programming, and Python provides powerful tools to perform various file operations. In this summary, we covered key concepts and code examples related to file handling in Python, including writing, appending, and copying files.

Author(s)

Akansha Yadav

Changelog

Date	Version	Changed by	Change Description
2023-09-27	1.0	Akansha Yadav	Created a reading file
2023-10-10	1.1	Steve Hord	QA pass with minor edits

•

Beginner's Guide to NumPy

Estimated Time : 10 Minutes

Objective:

In this reading, you'll learn:

- Basics of NumPy
- How to create NumPy arrays
- Array attributes and indexing
- Basic operations like addition and multiplication

What is NumPy?

NumPy, short for **Num**erical **Py**thon, is a fundamental library for numerical and scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays. NumPy serves as the foundation for many data science and machine learning libraries, making it an essential tool for data analysis and scientific research in Python.

Key aspects of NumPy in Python:

- **Efficient data structures**: NumPy introduces efficient array structures, which are faster and more memory-efficient than Python lists. This is

crucial for handling large data sets.

- **Multi-dimensional arrays**: NumPy allows you to work with multi-dimensional arrays, enabling the representation of matrices and tensors. This is particularly useful in scientific computing.
- **Element-wise operations**: NumPy simplifies element-wise mathematical operations on arrays, making it easy to perform calculations on entire data sets in one go.
- **Random number generation**: It provides a wide range of functions for generating random numbers and random data, which is useful for simulations and statistical analysis.
- **Integration with other libraries**: NumPy seamlessly integrates with other data science libraries like SciPy, Pandas, and Matplotlib, enhancing its utility in various domains.
- **Performance optimization**: NumPy functions are implemented in low-level languages like C and Fortran, which significantly boosts their performance. It's a go-to choice when speed is essential.

Installation

If you haven't already installed NumPy, you can do so using pip:

```
1
pip install numpy
```

Creating NumPy arrays

You can create NumPy arrays from Python lists. These arrays can be one-dimensional or multi-dimensional.

Creating 1D array

```
1
```

2. import numpy as np

import numpy as np: In this line, the NumPy library is imported and assigned an alias np to make it easier to reference in the code.

1. 1
2. 2
3. # Creating a 1D array
4. arr_1d = np.array([1, 2, 3, 4, 5]) # **np.array()** is used to create NumPy arrays.

arr_1d = np.array([1, 2, 3, 4, 5]): In this line, a one-dimensional NumPy array named arr_1d is created. It uses the np.array() function to convert a Python list [1, 2, 3, 4, 5] into a NumPy array. This array contains five elements, which are 1, 2, 3, 4, and 5. arr_1d is a 1D array because it has a single row of elements.

Creating 2D array

1. 1
2. import numpy as np

import numpy as np: In this line, the NumPy library is imported and assigned an alias np to make it easier to reference in the code.

1. 1
2. 2
3. # Creating a 2D array
4. arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]): In this line, a two-dimensional NumPy array named arr_2d is created. It uses the np.array() function to convert a list of lists into a 2D NumPy array.

The outer list contains three inner lists, each of which represents a row of elements. So, arr_2d is a 2D array with three rows and three columns. The elements in this array form a matrix with values from 1 to 9, organized in

a 3x3 grid.

Array attributes

NumPy arrays have several useful attributes:

```
1
2
3
4
5
6
7
# Array attributes
print(arr_2d.ndim)  # ndim : Represents the number of dimensions or "rank" of the array.
# output : 2
print(arr_2d.shape)  # shape : Returns a tuple indicating the number of rows and columns in the array.
# Output : (3, 3)
print(arr_2d.size) # size: Provides the total number of elements in the array.
# Output : 9
```

Indexing and slicing

You can access elements of a NumPy array using indexing and slicing:

In this line, the third element (index 2) of the 1D array arr_1d is accessed.

```
1
2
# Indexing and slicing
print(arr_1d[2])          # Accessing an element (3rd element)
```

In this line, the element in the 2nd row (index 1) and 3rd

column (index 2) of the 2D array arr_2d is accessed.

```
1
print(arr_2d[1, 2])       # Accessing an element (2nd row, 3rd column)
```

In this line, the 2nd row (index 1) of the 2D array arr_2d is accessed.

```
1
print(arr_2d[1])          # Accessing a row (2nd row)
```

In this line, the 2nd column (index 1) of the 2D array arr_2d is accessed.

```
1
print(arr_2d[:, 1])       # Accessing a column (2nd column)
```

Basic operations

NumPy simplifies basic operations on arrays:

Element-wise arithmetic operations:

Addition, subtraction, multiplication, and division of arrays with scalars or other arrays.

Array addition

```
1
2
3
4
5
# Array addition
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
result = array1 + array2
print(result)  # [5 7 9]
```

Scalar multiplication

```
1
2
3
```

```
4
# Scalar multiplication
array = np.array([1, 2, 3])
result = array * 2 # each element of an array is multiplied by 2
print(result)  # [2 4 6]
```

Element-wise multiplication (Hadamard Product)

```
1
2
3
4
5
# Element-wise multiplication (Hadamard product)
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
result = array1 * array2
print(result)  # [4 10 18]
```

Matrix multiplication

```
1
2
3
4
5
6
7
# Matrix multiplication
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
result = np.dot(matrix1, matrix2)
print(result)
# [[19 22]
```

14. # [43 50]]

NumPy simplifies these operations, making it easier and more efficient than traditional Python lists.

Operation with NumPy

Here's the list of operation which can be performed using Numpy

Operation	Description	Example
Array Creation	Creating a NumPy array.	arr = np.array([1, 2, 3, 4, 5])
Element-Wise Arithmetic	Element-wise addition, subtraction, and so on.	result = arr1 + arr2
Scalar Arithmetic	Scalar addition, subtraction, and so on.	result = arr * 2
Element-Wise Functions	Applying functions to each element.	result = np.sqrt(arr)
Sum and Mean	Calculating the sum and mean of an array.Calculating the sum and mean of an array.	total = np.sum(arr) average = np.mean(arr)

Maximum and Minimum Values	Finding the maximum and minimum values.	max_val = np.max(arr) min_val = np.min(arr)
Reshaping	Changing the shape of an array.	reshaped_arr = arr.reshape(2, 3)
Transposition	Transposing a multi-dimensional array.	transposed_arr = arr.T
Matrix Multiplication	Performing matrix multiplication.	result = np.dot(matrix1, matrix2)

Conclusion

NumPy is a fundamental library for data science and numerical computations. This guide covers the basics of NumPy, and there's much more to explore. Visit numpy.org for more information and examples.

Author

Akansha Yadav

Working with Data in Python Cheat Sheet

Reading and writing files

Package/ Method	Description	Syntax and Code Example
File opening modes	Different modes to open files for specific operations.	Syntax: r (reading) w (writing) a (appending) + (updating: read/ write) b (binary, otherwise text) 1 1 1 Examples: with open("data.txt", "r") as file: content = file.read() print(content) with open("output.txt", "w") as file: file.write("Hello, world!") with open("log.txt", "a") as file: file.write("Log entry: Something happened.") with open("data.txt",

		"r+") as file: content = file.read() file.write("Updated content: " + content)</td>
File reading methods	Different methods to read file content in various ways.	Syntax: 1 1 2 2 3 3 1 file.readlines() # reads all lines as a list 2 readline() # reads the next line as a string 3 file.read() # reads the entire file content as a string

Example:

```
1. 1
2. 2
3. 3
4. 4
5. with open("data.txt", "r") as file:
6.     lines = file.readlines()
7.     next_line = file.readline()
8.     content = file.read()
```

File writing methods	Different write methods to write content to a file.	Syntax: 1 1 2 2 1 file.write(content) # writes a string to the file 2 file.writelines(lines) # writes a list of strings to the file

Example:

```
1. 1
2. 2
3. 3
4. lines = ["Hello\n", "World\n"]
5. with open("output.txt", "w") as file:
6.     file.writelines(lines)
```

Iterating over lines	Iterates through each line in the file using a `loop`.	Syntax: 1 1 1 for line in file: # Code to process each line

Example:

```
1. 1
2. 2
3. with open("data.txt", "r") as file:
4. for line in file: print(line)
```

Open() and close()	Opens a file, performs operations, and explicitly closes the file using the close() method.	Syntax: 1 1 2 2 1 file = open(filename, mode) # Code that uses the file 2 file.close()

Example:

```
1. 1
2. 2
3. 3
4. file = open("data.txt", "r")
5. content = file.read()
6. file.close()
```

with open()	Opens a file using a with block, ensuring automatic file closure after usage.	Syntax: 1 1 1 with open(filename, mode) as file: # Code that uses the file

Example:

```
1. 1
2. 2
3. with open("data.txt", "r") as file:
4. content = file.read()
```

Pandas

Package/ Method	Description	Syntax and Code Example
.read_csv()	Reads data from a `.CSV` file and creates a DataFrame.	Syntax: dataframe_name = pd.read_csv("filename.csv") Example: df = pd.read_csv("data.csv")
.read_excel()	Reads data from an Excel file and creates a DataFrame.	Syntax: 1 1 1 dataframe_name = pd.read_excel("filename.xlsx")

Example:

```
1. 1
2. df = pd.read_excel("data.xlsx")
```

.to_csv()	Writes DataFrame to a CSV file.	Syntax: 1 1 1 dataframe_name.to_csv("output.csv", index=False)

Example:

```
1. 1
2. df.to_csv("output.csv", index=False)
```

Access Columns	Accesses a specific column using [] in the DataFrame.	Syntax: 1 1 2 2 1 dataframe_name["column_name"] # Accesses single column 2 dataframe_name[["column1", "column2"]] # Accesses multiple columns

Example:

```
1. 1
2. 2
3. df["age"]
4. df[["name", "age"]]
```

describe()	Generates statistics summary of numeric columns in the DataFrame.	Syntax: 1 1 1 dataframe_name.describe()

Example:

```
1. 1
2. df.describe()
```

drop()	Removes specified rows or columns from the DataFrame. axis=1 indicates columns. axis=0 indicates rows.	Syntax: 1 1 2 2 1 dataframe_name.drop(["column1", "column2"], axis=1, inplace=True) 2 dataframe_name.drop(index=[row1, row2], axis=0, inplace=True)

Example:

```
1. 1
2. 2
3. df.drop(["age", "salary"], axis=1, inplace=True) # Will drop columns
4. df.drop(index=[5, 10], axis=0, inplace=True) # Will drop rows
```

dropna()	Removes rows with missing NaN values from the DataFrame. axis=0 indicates rows.	Syntax: 1 1 1 dataframe_name.dropna(axis=0, inplace=True)

Example:

```
1. 1
2. df.dropna(axis=0, inplace=True)
```

duplicated()	Duplicate or repetitive values or records within a data set.	Syntax: 1 1 1 dataframe_name.duplicated()

Example:

```
1
duplicate_rows = df[df.duplicated()]
```

Filter Rows	Creates a new DataFrame with rows that meet specified conditions.	Syntax: 1 1 1 filtered_df = dataframe_name[(Conditional_statements)]

Example:

```
1
filtered_df = df[(df["age"] > 30) & (df["salary"] < 50000)
```

groupby()	Splits a DataFrame into groups based on specified criteria, enabling subsequent aggregation, transformation, or analysis within each group.	Syntax: 1 1 2 2 1 grouped = dataframe_name.groupby(by, axis=0, level=None, as_index=True, 2 sort=True, group_keys=True, squeeze=False, observed=False, dropna=True)

Example:

```
1. 1
2. grouped = df.groupby(["category",
   "region"]).agg({"sales": "sum"})
```

head()	Displays the first n rows of the DataFrame.	Syntax: 1 1 1 dataframe_name.head(n)

Example:

```
1. 1
2. df.head(5)
```

Import pandas	Imports the Pandas library with the alias pd.	Syntax: 1 1 1 import pandas as pd

Example:

```
1
import pandas as pd
```

info()	Provides information about the DataFrame, including data types and memory usage.	Syntax: 1 1 1 dataframe_name.info()

Example:

```
1
df.info()
```

merge()	Merges two DataFrames based on multiple common columns.	Syntax: 1 1 1 merged_df = pd.merge(df1, df2, on=["column1", "column2"])

Example:

```
1
merged_df = pd.merge(sales, products, on=["product_id", "category_id"])
```

print DataFrame	Displays the content of the DataFrame.	Syntax: 1 1 1 print(df) # or just type df

Example:

```
1
2
print(df)
df
```

replace()	Replaces specific values in a column with new values.	Syntax: 1 1 1 dataframe_name["column_name"].replace(old_value, new_value, inplace=True)

Example:

```
1
df["status"].replace("In Progress", "Active", inplace=True)
```

tail()	Displays the last n rows of the DataFrame.	Syntax: 1 1 1 dataframe_name.tail(n)

Example:

```
1
df.tail(5)
```

Numpy

Package/ Method	Description	Syntax and Code Example
Importing NumPy	Imports the NumPy library.	Syntax: 1 1 1 import numpy as np

Example:

```
1. 1
2. import numpy as np
```

np.array()	Creates a one or multi-dimensional array,	Syntax: 1 1 2 2 1 array_1d = np.array([list1 values]) # 1D Array 2 array_2d = np.array([[list1 values], [list2 values]]) # 2D Array

Example:

```
1. 1
2. 2
3. array_1d = np.array([1, 2, 3]) # 1D Array
4. array_2d = np.array([[1, 2], [3, 4]]) # 2D Array
```

Numpy Array Attributes	- Calculates the mean of array elements - Calculates the sum of array elements - Finds the minimum value in the array - Finds the maximum value in the array - Computes dot product of two arrays	Example: 1 1 2 2 3 3 4 4 5 5 1 np.mean(array) 2 np.sum(array) 3 np.min(array 4 np.max(array) 5 np.dot(array_1, array_2)

WEB SCRAPING

•

Web Scraping: A Key Tool in Data Science

Estimated Effort: 5 mins

Introduction

Web scraping, also known as web harvesting or web data extraction, is a technique used to extract large amounts of data from websites. The data on websites is unstructured, and web scraping enables us to convert it into a structured form.

Importance of Web Scraping in Data Science

In the field of data science, web scraping plays an integral role. It is used for various purposes such as:

1. **Data Collection:** Web scraping is a primary method of collecting data from the internet. This data can be used for analysis, research, etc.
2. **Real-time Application:** Web scraping is used for real-time applications like weather updates, price comparison, etc.
3. **Machine Learning:** Web scraping provides the data needed to train machine learning models.

Web Scraping with Python

Python provides several libraries for web scraping. Here are some of them:

1. **BeautifulSoup:** BeautifulSoup is a Python library used for web scraping purposes to pull the data out of HTML and XML files. It creates a parse tree from page source code that can be used to extract data in a hierarchical and more readable manner.
2. 1

3. 2
4. 3
5. 4
6. 5
7. from bs4 import BeautifulSoup
8. import requests
9. URL = "http://www.example.com"
10. page = requests.get(URL)
11. soup = BeautifulSoup(page.content, "html.parser")
12. **Scrapy:** Scrapy is an open-source and collaborative web crawling framework for Python. It is used to extract the data from the website.
13. 1
14. 2
15. 3
16. 4
17. 5
18. 6
19. 7
20. import scrapy
21. class QuotesSpider(scrapy.Spider):
22. name = "quotes"
23. start_urls = ['http://quotes.toscrape.com/tag/humor/',]
24. def parse(self, response):
25. for quote in response.css('div.quote'):
26. yield {'quote': quote.css('span.text::text').get()}
27. **Selenium:** Selenium is a tool used for controlling web browsers through programs and automating

browser tasks.

```
1
2
3
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://www.example.com")
```

Applications of Web Scraping

Web scraping is used in various fields and has many applications:

1. **Price Comparison:** Services such as ParseHub use web scraping to collect data from online shopping websites and use it to compare the prices of products.
2. **Email address gathering:** Many companies that use email as a medium for marketing, use web scraping to collect email ID and then send bulk emails.
3. **Social Media Scraping:** Web scraping is used to collect data from Social Media websites such as Twitter to find out what's trending.

Conclusion

Web scraping is an essential skill in the fast-growing world of data science. It provides the ability to turn the web into a source of data that can be analyzed, processed, and used for a variety of applications. However, it's important to remember that one should use web scraping responsibly and ethically, respecting the terms of use or robots.txt files of the websites being scraped.

Author(s)

Abhishek Gagneja

-

Web Scraping Tables using Pandas

Estimated Effort: 5 mins

The Pandas library in Python contains a function read_html() that can be used to extract tabular information from any web page.

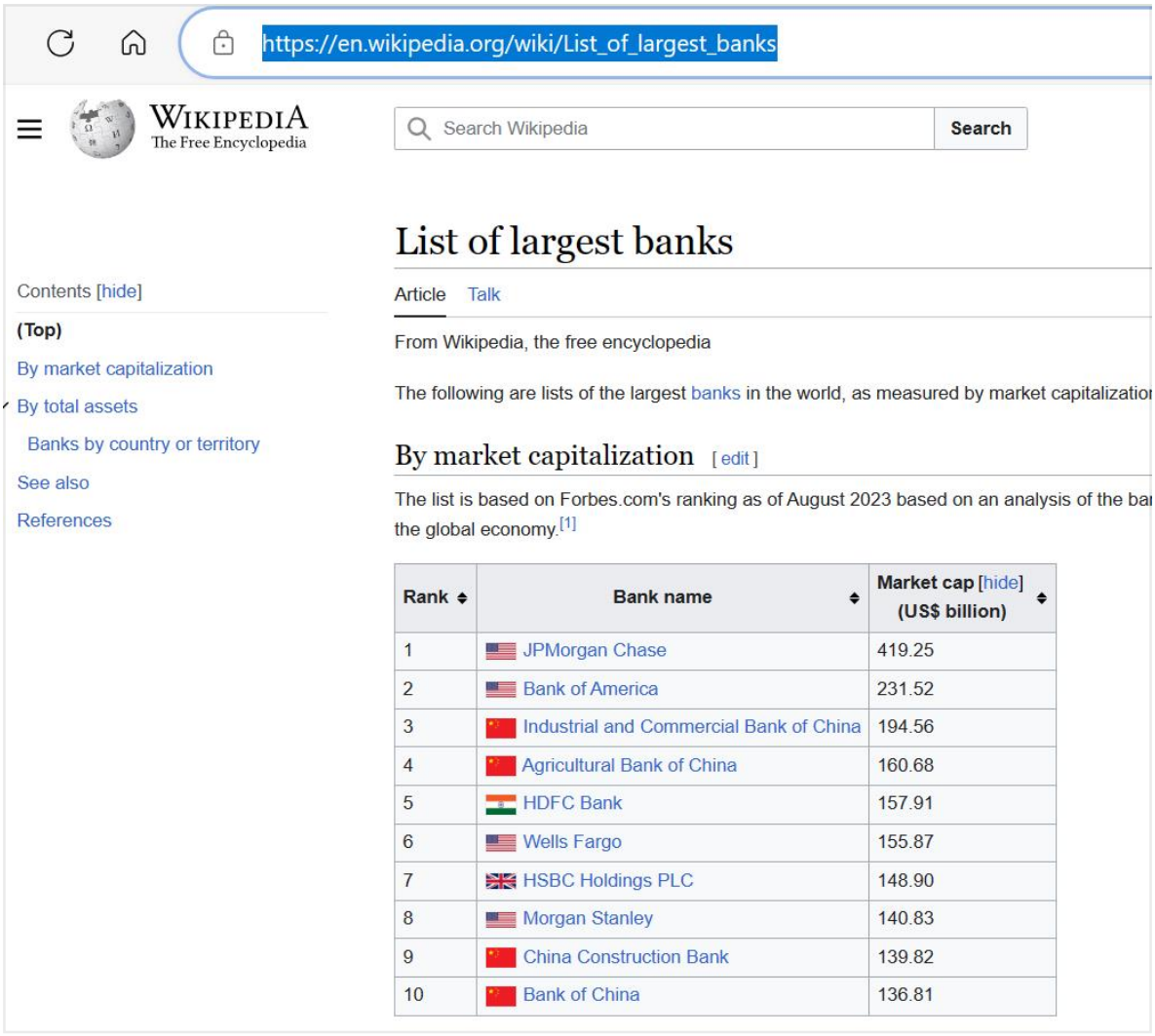
Consider the following example:

Let us assume we want to extract the list of the largest banks in the world by market capitalization, from the following link:

1. 1
2. URL = 'https://en.wikipedia.org/wiki/List_of_largest_banks'

We may use pandas.read_html() function in python to extract all the tables in the web page directly.

A snapshot of the webpage is shown below.



Rank	Bank name	Market cap [hide] (US$ billion)
1	JPMorgan Chase	419.25
2	Bank of America	231.52
3	Industrial and Commercial Bank of China	194.56
4	Agricultural Bank of China	160.68
5	HDFC Bank	157.91
6	Wells Fargo	155.87
7	HSBC Holdings PLC	148.90
8	Morgan Stanley	140.83
9	China Construction Bank	139.82
10	Bank of China	136.81

We can see that the required table is the first one in the web page.

Note: This is a live web page and it may get updated over time. The image shown above has been captured in November 2023. The process of data extraction remains the same.

We may execute the following lines of code to extract the required table from the web page.

1. 1
2. 2
3. 3
4. 4
5. 5
6. import pandas as pd
7. URL = 'https://en.wikipedia.org/wiki/

List_of_largest_banks'

8. tables = pd.read_html(URL)
9. df = tables[0]
10. print(df)

This will extract the required table as a dataframe df. The output of the print statement would look as shown below.

	Rank	Bank name	Market cap(US$ billion)
0	1	JPMorgan Chase	419.25
1	2	Bank of America	231.52
2	3	Industrial and Commercial Bank of China	194.56
3	4	Agricultural Bank of China	160.68
4	5	HDFC Bank	157.91
5	6	Wells Fargo	155.87
6	7	HSBC Holdings PLC	148.90
7	8	Morgan Stanley	140.83
8	9	China Construction Bank	139.82
9	10	Bank of China	136.81

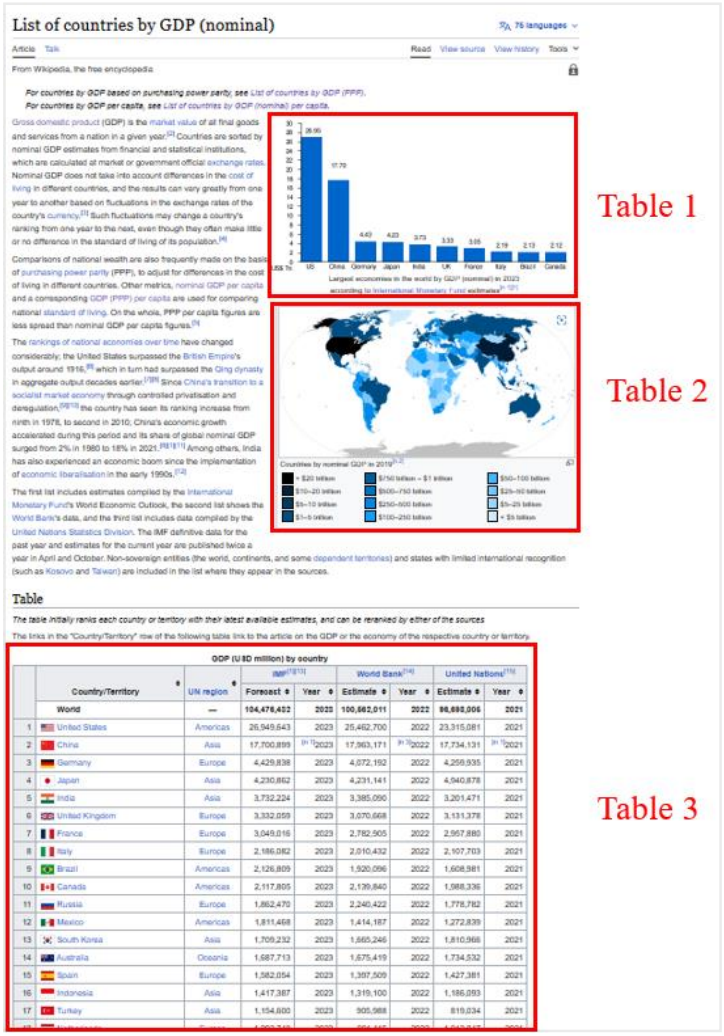
Although convenient, this method comes with its own set of limitations.

Firstly, web pages may have content saved in them as tables but they may not appear as tables on the web page.

For instance, consider the following URL showing the list of countries by GDP (nominal).

1. 1
2. URL = 'https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)'

The images on the web page are also saved in tabular format. A snapshot of the web page is shared below.



Secondly, the contents of the tables in the web pages may contain elements such as hyperlink text and other denoters, which are also scraped directly using the pandas method. This may lead to a requirement of further cleaning of data.

A closer look at table 3 in the image shown above indicates that there are many hyperlink texts which are also going to be treated as information by the pandas function.

GDP (USD million) by country

	Country/Territory	UN region	IMF[1][13]		World Bank[14]		United Nations[15]	
			Forecast	Year	Estimate	Year	Estimate	Year
	World	—	**104,476,432**	**2023**	**100,562,011**	**2022**	**96,698,005**	**2021**
1	United States	Americas	26,949,643	2023	25,462,700	2022	23,315,081	2021
2	China	Asia	17,700,899	[n 1]2023	17,963,171	[n 3]2022	17,734,131	[n 1]2021
3	Germany	Europe	4,429,838	2023	4,072,192	2022	4,259,935	2021
4	Japan	Asia	4,230,862	2023	4,231,141	2022	4,940,878	2021
5	India	Asia	3,732,224	2023	3,385,090	2022	3,201,471	2021
6	United Kingdom	Europe	3,332,059	2023	3,070,668	2022	3,131,378	2021
7	France	Europe	3,049,016	2023	2,782,905	2022	2,957,880	2021
8	Italy	Europe	2,186,082	2023	2,010,432	2022	2,107,703	2021
9	Brazil	Americas	2,126,809	2023	1,920,096	2022	1,608,981	2021
10	Canada	Americas	2,117,805	2023	2,139,840	2022	1,988,336	2021
11	Russia	Europe	1,862,470	2023	2,240,422	2022	1,778,782	2021
12	Mexico	Americas	1,811,468	2023	1,414,187	2022	1,272,839	2021
13	South Korea	Asia	1,709,232	2023	1,665,246	2022	1,810,966	2021
14	Australia	Oceania	1,687,713	2023	1,675,419	2022	1,734,532	2021
15	Spain	Europe	1,582,054	2023	1,397,509	2022	1,427,381	2021

We can extract the table using the code shown below.

1. 1
2. 2
3. 3
4. 4
5. 5
6. import pandas as pd
7. URL = 'https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)'
8. tables = pd.read_html(URL)
9. df = tables(2) # the required table will have index 2
10. print(df)

The output of the print statement is shown below.

	Country/Territory	UN region	IMF[1][13]		World Bank[14]		United Nations[15]	
	Country/Territory	UN region	Forecast	Year	Estimate	Year	Estimate	Year
0	World	–	104476432	2023	100562011	2022	96698005	2021
1	United States	Americas	26949643	2023	25462700	2022	23315081	2021
2	China	Asia	17700899	[n 1]2023	17963171	[n 3]2022	17734131	[n 1]2021
3	Germany	Europe	4429838	2023	4072192	2022	4259935	2021
4	Japan	Asia	4230862	2023	4231141	2022	4940878	2021
..	...	...	...	...	...	...	...	...
209	Palau	Oceania	267	2023	–	–	218	2021
210	Kiribati	Oceania	246	2023	223	2022	227	2021
211	Nauru	Oceania	150	2023	151	2022	155	2021
212	Montserrat	Americas	–	–	–	–	72	2021
213	Tuvalu	Oceania	63	2023	60	2022	60	2021

Note that the hyperlink texts have also been retained in the code output.
It is further prudent to point out, that this method exclusively operates only on tabular data extraction. BeautifulSoup library still remains the default method of extracting any kind of information from web pages.

Author(s)

Abhishek Gagneja

-

% buffered00:00
04:30

Cheat Sheet: API's and Data Collection

Package/ Method	Description	Code Example

Accessing element attribute	Access the value of a specific attribute of an HTML element.	Syntax: 1 1 1 attribute = element[(attribute)]

Example:

```
1. 1
2. href = link_element[(href)]
```

BeautifulSoup()	Parse the HTML content of a web page using BeautifulSoup. The parser type can vary based on the project.	Syntax: 1 1 1 soup = BeautifulSoup(html, (html.parser))

Example:

```
1. 1
2. html = (https://api.example.com/data) soup =
   BeautifulSoup(html, (html.parser))
```

delete()	Send a DELETE request to remove data or a resource from the server. DELETE requests delete a specified resource on the server.	Syntax: 1 1 1 response = requests.delete(url)

Example:

```
1. 1
```

2. response = requests.delete((https://
   api.example.com/delete))

find()	Find the first HTML element that matches the specified tag and attributes.	Syntax: 1 1 1 element = soup.find(tag, attrs)

Example:

1. 1
2. first_link = soup.find((a), {(class): (link)})

find_all()	Find all HTML elements that match the specified tag and attributes.	Syntax: 1 1 1 elements = soup.find_all(tag, attrs)

Example:

1. 1
2. all_links = soup.find_all((a), {(class): (link)})</td>

findChildren()	Find all child elements of an HTML element.	Syntax: 1 1 1 children = element.findChildren()

Example:

1. 1
2. child_elements = parent_div.findChildren()

get()	Perform a GET request to retrieve data from a specified URL. GET requests are typically used for reading data from an API. The response variable will contain the server's response, which you can process further.	Syntax: 1 1 1 response = requests.get(url)

Example:

```
1. 1
2. response = requests.get((https://
   api.example.com/data))
```

Headers	Include custom headers in the request. Headers can provide additional information to the server, such as authentication tokens or content types.	Syntax: 1 1 1 headers = {(HeaderName) : (Value)}

Example:

1. 1
2. base_url = (https://api.example.com/data) headers = {(Authorization): (Bearer YOUR_TOKEN)} response = requests.get(base_url, headers=headers)

Import Libraries	Import the necessary Python libraries for web scraping.	Syntax: 1 1 1 from bs4 import BeautifulSoup

json()	Parse JSON data from the response. This extracts and works with the data returned by the API. The response.json() method converts the JSON response into a Python data structure (usually a dictionary or list).	Syntax: 1 1 1 data = response.json()

Example:

```
1
2
response = requests.get((https://api.example.com/data))
data = response.json()
```

next_sibling()	Find the next sibling element in the DOM.	Syntax: 1 1 1 sibling = element.find_next_sibling()

Example:

```
1
next_sibling = current_element.find_next_sibling()
```

parent	Access the parent element in the Document Object Model (DOM).	Syntax: 1 1 1 parent = element.parent

Example:

```
1. 1
2. parent_div = paragraph.parent
```

post()	Send a POST request to a specified URL with data. Create or update POST requests using resources on the server. The data parameter contains the data to send to the server, often in JSON format.	Syntax: 1 1 1 response = requests.post(url, data)

Example:

```
1. 1
2. response = requests.post((https://
   api.example.com/submit), data={(key): (value)})
```

put()	Send a PUT request to update data on the server. PUT requests are used to update an existing resource on the server with the data provided in the data parameter, typically in JSON format.	Syntax: 1 1 1 response = requests.put(url, data)

Example:

```
1. 1
2. response = requests.put((https://
   api.example.com/update), data={(key): (value)})
```

Query parameters	Pass query parameters in the URL to filter or customize the request. Query parameters specify conditions or limits for the requested data.	Syntax: 1 1 1 params = {(param_name): (value)}

Example:

```
1. 1
2. 2
```

```
3. 3
4. base_url = "https://api.example.com/data"
5. params = {"page": 1, "per_page": 10}
6. response = requests.get(base_url,
   params=params)
```

select()	Select HTML elements from the parsed HTML using a CSS selector.	Syntax: 1 1 1 element = soup.select(selector)

Example:

```
1. 1
2. titles = soup.select((h1))
```

status_code	Check the HTTP status code of the response. The HTTP status code indicates the result of the request (success, error, redirection). Use the HTTP status codeIt can be used for error handling and decision-making in your code.	Syntax: 1 1 1 response.status_code

Example:

1. 1
2. 2
3. 3
4. url = "https://api.example.com/data"
5. response = requests.get(url)
6. status_code = response.status_code

tags for find() and find_all()	Specify any valid HTML tag as the tag parameter to search for elements of that type. Here are some common HTML tags that you can use with the tag parameter.	Tag Example: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 1 - (a): Find anchor () tags. 2 - (p): Find paragraph ((p)) tags. 3 - (h1), (h2), (h3), (h4), (h5), (h6): Find heading tags from level 1 to 6 ((h1),n (h2)). 4 - (table): Find table () tags.

		5 - (tr): Find table row () tags. 6 - (td): Find table cell ((td)) tags. 7 - (th): Find table header cell ((td))tags. 8 - (img): Find image ((img)) tags. 9 - (form): Find form ((form)) tags. 10 - (button): Find button ((button)) tags.
text	Retrieve the text content of an HTML element.	Syntax: 1 1 1 text = element.text

Example:

```
1. 1
2. title_text = title_element.text
```