**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 2. 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 4. my\_list = [1, 2, 3, 4, 5] 5. new\_list = my\_list.copy() print(new\_list) 6. # 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 4. my\_list = [1, 2, 2, 3, 4, 2, 5, 2] 5. count = my\_list.count(2) print(count) 6. # 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 2. 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 4. my\_list = [10, 20, 30, 40, 50] 5. del my\_list[2] # Removes the element at index 2 print(my\_list) 6. # 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 2. list\_name.extend(iterable) |

Example:

1. 1
2. 2
3. 3
4. 4
5. fruits = ["apple", "banana", "orange"]
6. more\_fruits = ["mango", "grape"]
7. fruits.extend(more\_fruits)
8. 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 6. my\_list = [10, 20, 30, 40, 50] 7. print(my\_list[0]) 8. # Output: 10 (accessing the first element) 9. print(my\_list[-1]) 10. # Output: 50 (accessing the last element using negative indexing) |
| insert() | The `insert()` method is used to insert an element. | Syntax:   1. 1 2. 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 5. my\_list = [10, 20, 30, 40, 50] 6. my\_list[1] = 25 # Modifying the second element 7. print(my\_list) 8. # 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 8. my\_list = [10, 20, 30, 40, 50] 9. removed\_element = my\_list.pop(2) # Removes and returns the element at index 2 10. print(removed\_element) 11. # Output: 30 12. print(my\_list) 13. # 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. print(my\_list)
13. # 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 5. my\_list = [10, 20, 30, 40, 50] 6. my\_list.remove(30) # Removes the element 30 7. print(my\_list) 8. # 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 4. my\_list = [1, 2, 3, 4, 5] 5. my\_list.reverse() print(my\_list) 6. # Output: [5, 4, 3, 2, 1] |
| Slicing | You can use slicing to access a range of elements from a list. | Syntax:   1. 1 2. 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. print(my\_list[:3])
17. # Output: [1, 2, 3] (elements from the beginning up to index 2)
18. print(my\_list[2:])
19. # Output: [3, 4, 5] (elements from index 2 to the end)
20. print(my\_list[::2])
21. # 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 5. my\_list = [5, 2, 8, 1, 9] 6. my\_list.sort() 7. print(my\_list) 8. # 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 2. 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 2. 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 2. 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 2. 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 3. dict\_name = {} #Creates an empty dictionary 4. 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 2. 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 2. 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 3. if "name" in person: 4. 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 2. 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 2. 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 2. 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 2. 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 2. 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 2. 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 3. empty\_set = set() #Creating an Empty Set 4. 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 2. 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 2. 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 2. 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 2. 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 2. 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 5. union\_set = set1.union(set2) 6. intersection\_set = set1.intersection(set2) 7. difference\_set = set1.difference(set2) 8. 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 2. set\_name.update(iterable) |

Example:

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

CONDITION AND BRANCHING

LOOP

n this video we will cover Loops in particular for loops and while loops.

We will use many visual examples in this video.

See the labs for examples with data.

Before we talk about loops,

let's go over the range function.

The range function outputs and ordered sequence as a list I.

If the input is a positive integer,

the output is a sequence.

The sequence contains the same number of elements as the input but starts at zero.

For example, if the input is three the output is the sequence zero, one, two.

If the range function has two inputs where

the first input is smaller than the second input,

the output is a sequence that starts at the first input.

Then the sequence iterates up to but not including the second number.

For the input 10 and 15 we get the following sequence.

See the labs for more capabilities of the range function.

Please note, if you use Python three,

the range function will not generate a list explicitly like in Python two.

In this section, we will cover for loops.

We will focus on lists, but many of the procedures can be used on tuples.

Loops perform a task over and over.

Consider the group of colored squares.

Let's say we would like to replace each colored square with a white square.

Let's give each square a number to make things a little

easier and refer to all the group of squares as squares.

If we wanted to tell someone to replace squares zero with

a white square, we would say equals replace square

zero with a white square or we can say four squares

zero in squares square zero equals white square.

Similarly, for the next square we can say for square one in squares,

square one equals white square.

For the next square we can say for square two in squares,

square two equals white square.

We repeat the process for each square.

The only thing that changes is the index of the square we are referring to.

If we're going to perform a similar task in Python we cannot use actual squares.

So let's use a list to represent the boxes.

Each element in the list is a string representing the color.

We want to change the name of the color in each element to white.

Each element in the list has the following index.

This is a syntax to perform a loop in Python.

Notice the indent, the range function generates a list.

The code will simply repeat everything in the indent five times.

If you were to change the value to six it would do it 6 times.

However, the value of I is incremented by one each time.

In this segment we change the I element of the list to the string white.

The value of I is set to zero.

Each iteration of the loop starts at the beginning of the indent.

We then run everything in the indent.

The first element in the list is set to white.

We then go to the start of the indent,

we progress down each line.

When we reach the line to change the value of the list,

we set the value of index one to white.

The value of I increases by one.

We repeat the process for index two.

The process continues for the next index,

until we've reached the final element.

We can also iterate through a list or tuple directly in python,

we do not even need to use indices.

Here is the list squares.

Each iteration of the list we pass

one element of the list squares to the variable square.

Lets display the value of the variable square on this section.

For the first iteration,

the value of square is red,

we then start the second iteration.

For the second iteration,

the value of square is yellow.

We then start the third iteration. For the final iteration, the value of square is green,

a useful function for iterating data is enumerate.

It can be used to obtain the index and the element in the list.

Let's use the box analogy with the numbers representing the index of each square.

This is the syntax to iterate through a list and provide the index of each element.

We use the list squares and use the names of the colors to represent the colored squares.

The argument of the function enumerate is the list.

In this case squares the variable I is

the index and the variable square is the corresponding element in the list.

Let's use the left part of the screen to

display the different values of the variable square

and I for the various iterations of the loop. For the first iteration,

the value of the variable is red corresponding to

the zeroth index, and the value for I is zero for the second iteration.

The value of the variable square is yellow, and

the value of I corresponds to its index i.e.

1. We repeat the process for the last index.

While loops are similar to for loops but instead of executing

a statement a set number of times a while loop will only run if a condition is met.

Let's say we would like to copy

all the orange squares from the list squares to the list New squares.

But we would like to stop if we encounter a non-orange square.

We don't know the value of the squares beforehand.

We would simply continue the process while the square is

orange or see if the square equals orange.

If not, we would stop. For the first example,

we would check if the square was orange.

It satisfies the conditions so we would copy the square.

We repeat the process for the second square. The condition is met.

So we copy the square. In the next iteration, we encounter a purple square.

The condition is not met.

So we stop the process.

This is essentially what a while loop does.

Let's use the figure on the left to represent the code. We will

use a list with the names of the color to represent the different squares.

We create an empty list of new squares.

In reality the list is of indeterminate size.

We start the index at zero the while statement will repeatedly

execute the statements within the indent until the condition inside the bracket is false.

We append the value of the first element of the list squares to the list new squares.

We increase the value of I by one.

We append the value of the second element of the list squares to the list new squares.

We increment the value of I.

Now the value in the array squares is purple;

therefore, the condition for the while statement is false and we exit the loop.

Check out the labs for more examples of loop many with real data.

(Music)