**WEEK 3**

**In Python, you can define functions to perform operations on data structures, such as lists, dictionaries, and sets. Functions are a way to encapsulate a specific set of operations or algorithms that can be applied to data structures.**

1. List Functions

- Adding Elements to a List

def add\_element(my\_list, element):

my\_list.append(element)

Removing Elements from a List

def remove\_element(my\_list, element):

my\_list.remove(element)

Sorting a List

python

def sort\_list(my\_list):

my\_list.sort()

-Filtering a List

python

def filter\_list(my\_list, condition):

return [item for item in my\_list if condition(item)]

2. Dictionary Functions

-Adding Key-Value Pairs to a Dictionary

def add\_to\_dict(my\_dict, key, value):

my\_dict[key] = value

Removing Key-Value Pairs from a Dictionary

python

def remove\_from\_dict (my\_dict, key):

del my\_dict[key]

Accessing Values in a Dictionary

python

def get\_value(my\_dict, key, default\_value=None):

return my\_dict.get(key, default\_value)

Iterating Over a Dictionary

python

def iterate\_dict(my\_dict):

for key, value in my\_dict.items():

print(f"Key: {key}, Value: {value}")

3. Set Functions

-Adding Elements to a Set

python

def add\_to\_set(my\_set, element):

my\_set.add(element)

Removing Elements from a Set

ppython

def remove\_from\_set(my\_set, element):

my\_set.remove(element)

Set Operations (Union, Intersection, Difference)

```python

def set\_operations(set1, set2):

union = set1.union(set2)

intersection = set1.intersection(set2)

difference = set1.difference(set2)

**more on lists**

**list.append(x)**

Add an item to the end of the list. Equivalent to a[len(a):] = [x].

**list.extend(iteration)**

Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

**list.insert(I, x)**

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

**list,remove(x)**

Remove the first item from the list whose value is equal to x. It raises a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \t "_blank) if there is no such item.

**list.pop([i])**

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

**list.clear()**

Remove all items from the list. Equivalent to del a[:].

**list.index(x[,start[, end]])**

Return zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item.

The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

**list.count(x)**

Return the number of times x appears in the list.

**list.sort(key=None, reverse=false=false)**

Sort the items of the list in place (the arguments can be used for sort customization, see sort() for their explanation).

**list.reverse()**

Reverse the elements of the list in place.

Return a shallow copy of the list. Equivalent to a[:].

**Using Lists as Stacks and Queues**

Lists can be nested to form sophisticated data structures and hold components of various types. You can use this flexibility to create structured representations of a variety of data types.

Lists allow you to change, add, or remove elements to alter the content of the list. They are therefore appropriate in scenarios where data must be updated or changed over time.

 Addition, deletion, sorting, and element searching are just a few of the many built-in operations that lists may do. Working with lists is made convenient and efficient by these techniques.

 List comprehension in Python are a succinct and legible approach to make new lists by giving each item in a list an expression.

Using lists as stacks and queues in Python is a practical way to manage data structures that adhere to the Last-In-First-Out (LIFO) and First-In-First-Out (FIFO) principles, respectively.

Using Lists as Stacks

A stack is a data structure where the last element added is the first one to be removed You can use Python lists to implement a stack

You can use the `append()` method to push elements onto the stack.

python

stack = []

stack.append(1)

stack.append(2)

stack.append(3)

-Pop (Remove from the stack): Use the `pop()` method to remove elements from the top of the stack.

python

top\_element = stack.pop() # Removes and returns the top element

```

-View the Top Element: To view the top element without removing it, you can use indexing.

python

top\_element = stack[-1] # View the top element without removing it

Using Lists as Queues (FIFO)

A queue is a data structure where the first element added is the first one to be removed (First-In-First-Out or FIFO). While you can use lists, it's more efficient to use Python's `collections.deque` for implementing a queue. Here are the basic operations:

- nqueue (Add to the queue): Use the `append()` method to add elements at the rear of the queue.

```python

from collections import deque

queue = deque()

queue.append(1)

queue.append(2)

queue.append(3)

-Dequeue (Remove from the queue): Use the `popleft()` method to remove elements from the front of the queue.

front\_element = queue.popleft() # Removes and returns the front element

View the Front Element To view the front element without removing it, use indexing.

front\_element = queue[0] # View the front element without removing it

**Looping techniques**

in programming are methods for iterating through data structures (e.g., lists, tuples, dictionaries) or sequences (e.g., strings) to perform specific tasks efficiently and with clear code.

1. For Loops For loops are used to iterate through an iterable (e.g., a list) and execute a block of code for each item in the iterable.

for item in iterable:

# code to execute for each item

2. While Loops While loops repeatedly execute a block of code as long as a specified condition is met.

while condition:

# code to execute if the condition is True

3. Enumerate The `enumerate` function is used to iterate through an iterable while keeping track of the index or position of each item.

for index, item in enumerate(iterable):

# code to execute with index and item

4. ZipThe `zip` function is used to iterate through multiple iterables in parallel, combining items from each iterable.

for item1, item2 in zip(iterable1, iterable2):

# code to execute for items from both iterables

5. Range The `range` function generates a sequence of numbers that can be used for looping, often in `for` loops.

for i in range(start, stop, step):

# code to execute for each value of i

6. Iterating Over Dictionaries

- Looping through keys:

for key in dictionary:

# code to execute for each key

- Looping through values:

for value in dictionary.values():

# code to execute for each value

- Looping through key-value pairs (items):

for key, value in dictionary.items():

# code to execute for each key-value pair

7. List Comprehensions List comprehensions provide a concise way to create new lists by applying an expression to each item in an iterable.

python

new list = [expression for item in iterable]

8. While-Else and For-Else Python allows the use of an `else` block after a `while` or `for` loop, which is executed when the loop finishes normally (not due to a `break` statement).

9. Nested Loops You can nest loops within each other to perform iterations within iterations, useful for working with multi-dimensional data structures.10. Break and Continue The `break` statement is used to exit a loop prematurely, while the `continue` statement is used to skip the current iteration and continue to the next.

**More on conditions**

Conditions, often referred to as conditional statements, are fundamental constructs in programming that allow you to control the flow of your code based on certain conditions or criteria. In Python, you can use various conditional statements and operators to make decisions and execute specific code blocks. Here's more on conditions in Python:

1. \*\*if Statement\*\*: The `if` statement is used to execute a block of code if a specified condition is true. It can be followed by optional `elif` (short for "else if") and `else` blocks.

```python

if condition:

# code to execute if condition is true

elif another\_condition:

# code to execute if another condition is true

else:

# code to execute if no conditions are true

```

2. \*\*Comparison Operators\*\*: Python provides a range of comparison operators to compare values. Common comparison operators include `==` (equal), `!=` (not equal), `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal).

3. \*\*Logical Operators\*\*: Logical operators are used to combine multiple conditions. Common logical operators include `and`, `or`, and `not`.

```python

if condition1 and condition2:

# code to execute if both conditions are true

if condition1 or condition2:

# code to execute if at least one condition is true

if not condition:

# code to execute if the condition is false

```

4. \*\*Nested if Statements\*\*: You can nest `if` statements within other `if` statements to create more complex decision-making structures.

5. \*\*Ternary Conditional Operator\*\*: Python allows for a one-liner conditional expression (ternary operator) for simple cases.

```python

value = x if condition else y

```

6. \*\*Switch-Case (Dictionary Mapping)\*\*: Python does not have a traditional switch-case statement, but you can emulate it using dictionaries to map values to corresponding actions.

7. \*\*Truthy and Falsy Values\*\*: In Python, some values are considered "truthy" (evaluate to `True`) and others are "falsy" (evaluate to `False`) when used in conditions. For example, an empty list or string is considered falsy, while non-empty ones are truthy.

8. \*\*Short-Circuit Evaluation\*\*: Python uses short-circuit evaluation with logical operators. If the first part of a logical `and` statement is `False`, the second part is not evaluated. Similarly, if the first part of a logical `or` statement is `True`, the second part is not evaluated.

9. \*\*Identity Operators\*\*: Python provides the `is` and `is not` operators to check if two variables reference the same object in memory.

```python

if x is y:

# code to execute if x and y refer to the same object

```

10. \*\*In Operator\*\*: The `in` operator is used to check if a value exists in an iterable (e.g., a list, tuple, or string).

```python

if item in iterable:

# code to execute if item is found in iterable

```

11. \*\*Complex Conditions\*\*: You can create complex conditions by combining multiple conditions and operators to make intricate decisions.

12. \*\*Error Handling with try-except\*\*: The `try` and `except` blocks are used to handle exceptions and errors in Python code. They allow you to gracefully handle unexpected situations.

```python

try:

# code that might raise an exception

except ExceptionType:

# code to handle the exception

```

Conditional statements are crucial for creating decision-making processes in your programs, allowing your code to respond dynamically to changing circumstances or data. By understanding and using conditions effectively, you can control the behavior of your programs and make them more versatile and robust.