**Python Day 3: Operations on Sequences**

**1. Concatenation (The + Operator)**

Concatenation is the process of joining two or more sequences end-to-end to create a new, single sequence. Think of it like linking two chains together.

**Concatenating Strings (str)**

When you use the + operator with strings, you join them to form a new, longer string.

**Code Example 1: Simple Joining**

a='Raj'

b='Kadam'

print(a+b)

*Output:*

RajKadam

**Code Example 2: Joining with a Separator**

a='Raj'

b='Kadam'

print(a+" "+b)

print(a+" @ "+b)

*Output:*

Raj Kadam

Raj @ Kadam

**Explanation:**

* print(a+" "+b): Here, we concatenate three strings: a, a literal space string " ", and b. This is the standard way to combine parts of a name or sentence.
* print(a+" @ "+b): This shows that you can join strings with any other string as a separator, in this case " @ ".

**Concatenating Tuples (tuple)**

The + operator works similarly for tuples, creating a new tuple that contains all the elements from the original tuples.

**Code Example**

x=(1,2)

y=(3,4)

print(x+y)

*Output:*

(1, 2, 3, 4)

**Explanation:**

* The elements of tuple y are added to the end of tuple x, resulting in a new, single tuple.

**Concatenating Lists (list)**

Lists can also be concatenated to combine their elements into a new list.

**Code Example**

m=['R','B']

l=['Q','S']

print(l+m)

print(l+[])

print(l+[1,2]+m)

*Output:*

['Q', 'S', 'R', 'B']

['Q', 'S']

['Q', 'S', 1, 2, 'R', 'B']

**Explanation:**

1. print(l+m): The list m is joined to the end of list l.
2. print(l+[]): This demonstrates that concatenating an empty list [] doesn't change the original list.
3. print(l+[1,2]+m): This shows you can chain multiple concatenations in one line to combine several lists and literal list values.

**2. Replication (The \* Operator)**

Replication is the process of repeating a sequence a certain number of times. It's like making multiple copies of the same item.

**Replicating Strings (str)**

You can replicate a string by multiplying it by an integer.

**Code Example**

w='R'

print(w\*3)

print('@'+w\*3+'@')

print((w+" ")\*3)

*Output:*

RRR

@RRR@

R R R

**Explanation:**

1. print(w\*3): The string 'R' is repeated 3 times, resulting in 'RRR'.
2. print('@'+w\*3+'@'): This shows how replication (w\*3) can be combined with concatenation (+) to build a more complex string.
3. print((w+" ")\*3): Here, the sequence being replicated is (w+" "), which is 'R '. This entire sequence is repeated 3 times.

**Replicating Tuples (tuple)**

Tuples can be replicated in the same way as strings.

**Code Example**

Python

z=(1,2)

print(z\*2)

print((z+('r','b'))\*2)

print((z+(2))\*2)

*Output:*

(1, 2, 1, 2)

(1, 2, 'r', 'b', 1, 2, 'r', 'b')

TypeError: can only concatenate tuple (not "int") to tuple

**Explanation:**

1. print(z\*2): The tuple z is repeated twice, creating a new tuple with the elements (1, 2, 1, 2).
2. print((z+('r','b'))\*2): This first concatenates z with ('r','b') to get (1, 2, 'r', 'b') and then replicates that new tuple twice.
3. print((z+(2))\*2): This line causes a **TypeError**.
   * **Why the error?** Python only knows how to concatenate a tuple with *another tuple*. It does not know how to add an integer (2) to a tuple.
   * **How to fix it:** To create a single-element tuple, you must add a trailing comma: (2,). The corrected code would be print((z+(2,))\*2).

**Replicating Lists (list)**

d=[1,1.1]

print((d+[1.1])\*3)

print(((d+[2])\*2)\*2)

*Output:*

[1, 1.1, 1.1, 1, 1.1, 1.1, 1, 1.1, 1.1]

[1, 1.1, 2, 1, 1.1, 2, 1, 1.1, 2, 1, 1.1, 2]

**Explanation:**

1. (d+[1.1])\*3: First, the list [1.1] is concatenated to d, resulting in [1, 1.1, 1.1]. Then, this new list is replicated 3 times.
2. ((d+[2])\*2)\*2: This is a nested operation.
   * Inside the innermost parentheses, d+[2] becomes [1, 1.1, 2].
   * This result is replicated by 2, becoming [1, 1.1, 2, 1, 1.1, 2].
   * Finally, this entire list is replicated by 2 again.

**3. Indexing (The [] Operator)**

Indexing is how you access a single element within a sequence using its position. Think of it like finding a book on a shelf by its number.

Key Concept: Zero-Based Indexing

In Python, counting starts from 0.

* The first element is at index 0.
* The second element is at index 1.
* And so on...

**Indexing Strings (str)**

You can access individual characters of a string by their index.

Python

s="shreya jadhav"

# s h r e y a j a d h a v

# 0 1 2 3 4 5 6 7 8 9 10 11 12

print(s[0]+s[7])

print(s[9]+s[1]+s[3]+s[3]+s[2]+s[5])

*Output:*

sj

dheera

**Explanation:**

* print(s[0]+s[7]): This fetches the character at index 0 ('s') and the character at index 7 ('j') and concatenates them.
* The second print statement retrieves characters from various positions in s and combines them to form a new word, 'dheera'.

**Indexing Tuples and Lists**

Indexing works the same way for tuples and lists. If a sequence contains other sequences (like a list of strings), you can use multiple indexes to access elements within the nested items.

**Code Example: Tuple with Nested Indexing**

t=('ram','maruti','road')

print((t[0][0],t[0][2],t[2][1]))

*Output:*

('r', 'm', 'o')

**Explanation:**

* t[0]: Accesses the first element of the tuple, which is the string 'ram'.
* t[0][0]: Accesses the character at index 0 *of the string 'ram'*, which is 'r'.
* t[0][2]: Accesses the character at index 2 *of the string 'ram'*, which is 'm'.
* t[2][1]: Accesses the element at index 2 ('road') and then the character at index 1 of that string ('o').

**Code Example: List with Mixed Types and Nested Indexing**

k=[1,2,3+4j,'raj']

print((k[0],k[3][0],k[1],k[3][1],k[2],k[3][2]))

*Output:*

(1, 'r', 2, 'a', (3+4j), 'j')

Explanation:

This example builds a new tuple by picking items from the list k.

* k[0] is the integer 1.
* k[3] is the string 'raj'. Therefore, k[3][0] is the first character, 'r'.
* This continues, pulling elements from the list and characters from the nested string 'raj'.

**4. Type Casting (Converting Between Data Types)**

Type casting (or type conversion) is the process of converting a value from one data type to another. Python provides simple functions for this, like list(), tuple(), set(), and str().

**Code Examples & Explanation**

**Example 1: str -> tuple -> list**

a='raj'

c=list(tuple(a))

print(c)

*Output:*

['r', 'a', 'j']

**Breakdown:**

1. tuple(a) converts the string 'raj' into a tuple ('r', 'a', 'j').
2. list(...) then converts that tuple into a list ['r', 'a', 'j'].

**Example 2: Using set() to Remove Duplicates**

b='veer'

c=list(set(tuple(b)))

print(c)

*Output:*

['v', 'r', 'e']

**Breakdown:**

1. tuple(b) converts 'veer' to ('v', 'e', 'e', 'r').
2. set(...) converts the tuple to a set. Sets only store unique values, so it becomes {'v', 'e', 'r'} (the order may vary).
3. list(...) converts the set into a list.

**Example 3: Casting a Dictionary**

x={'a':100,'b':200}

z=list(tuple(x))

print(z)

*Output:*

['a', 'b']

**Breakdown:**

* When you cast a dictionary to another sequence type, only the **keys** are used. tuple(x) becomes ('a', 'b'), which is then converted to the list ['a', 'b'].

**Example 4: Casting an int to set of characters**

m=121

n=str(m) # Convert integer 121 to string '121'

l=tuple(n) # Convert string '121' to tuple ('1', '2', '1')

p=set(l) # Convert tuple to set {'1', '2'} to get unique characters

print(p)

*Output:*

{'2', '1'}

**Breakdown:** This shows a multi-step conversion to find the unique digits in a number. The number must first be converted to a string before it can be treated as a sequence of characters.