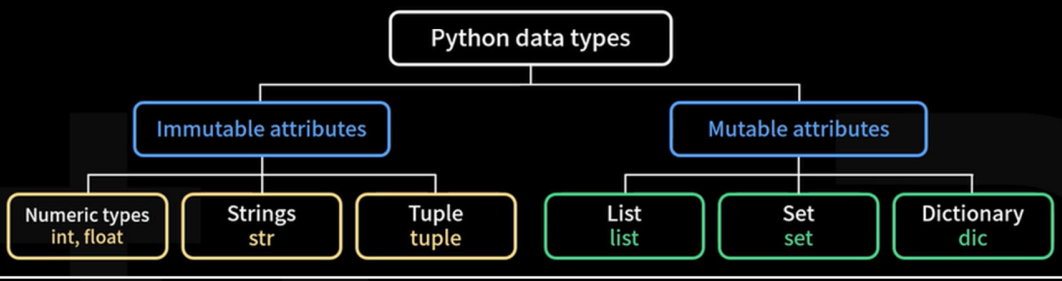
Tuple Data Type and Sequence Data Type

Topic 03 Key concept

1. Tuple

1.1 Immutable attribute, mutable attribute

- When classifying Python data types, we can divide them into immutable attributes and mutable attributes, as shown in the following diagram.



1. Tuple

1.1 Immutable attribute, mutable attribute

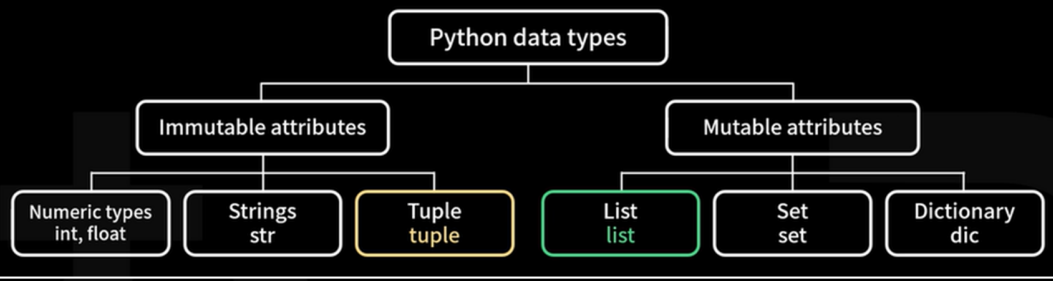
- When classifying Python data types, we can divide them into immutable a
mutable attributes, as shown in the following diagram.

Python has various data types, so you can choose and utilize them according to your purpose.

1. Tuple

1.1 Immutable attribute, mutable attribute

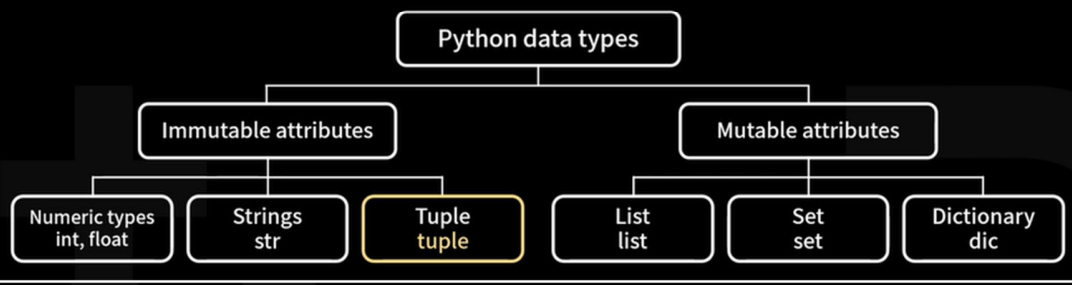
- A tuple is very similar to a list. However, the contents of a tuple cannot be changed once they are specified, making it a representative immutable data type.



1. Tuple

1.1 Immutable attribute, mutable attribute

- Since the contents of a tuple cannot be changed once they are determined, tuples have a simple structure and faster access speed compared to lists. Therefore, these two data types are selected and used depending on the purpose.



1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

- Let's create a simple tuple as follows.

```
1 colors = ("red", "green", "blue")
2 colors
```

```
('red', 'green', 'blue')
```

```
1 numbers = (1, 2, 3, 4, 5)
2 numbers
```

```
(1, 2, 3, 4, 5)
```

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

- Let's create a simple tuple as follows.

```
1 colors = ("red", "green", "blue")
2 colors
```

```
('red', 'green', 'blue')
```

Tuple for storing colors

```
1 numbers = (1, 2, 3, 4, 5)
2 numbers
```

```
(1, 2, 3, 4, 5)
```

Tuple for storing integer sequences

- It looks similar to a list. What are the differences?

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

TypeError

```
1 t1 = (1, 2, 3, 4, 5)
```

A tuple is an immutable data type. Therefore, be careful when trying to modify the elements of a tuple, as it will result in an error.

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

TypeError

```
1 t1 = (1, 2, 3, 4, 5)
2 t1[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 t1 = (1, 2, 3, 4, 5)
----> 2 t1[0] = 100

TypeError: 'tuple' object does not support item assignment
```

AnB_Prime

---><---

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

TypeError

```
t1 = (1, 2, 3, 4, 5)
t1[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 t1 = (1, 2, 3, 4, 5)
----> 2 t1[0] = 100

TypeError: 'tuple' object does not support item assignment
```

- The most important characteristic of a tuple is that "its values do not change".

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

TypeError

```
t1 = (1, 2, 3, 4, 5)
t1[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 t1 = (1, 2, 3, 4, 5)
----> 2 t1[0] = 100

TypeError: 'tuple' object does not support item assignment
```

- If you try to change the elements of a tuple as in the following code, a TypeError will occur.

1. Tuple

1.2 Immutable data type, which cannot be modified once created: Tuple

TypeError

```
t1 = (1, 2, 3, 4, 5)
t1[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 t1 = (1, 2, 3, 4, 5)
----> 2 t1[0] = 100

TypeError: 'tuple' object does not support item assignment
```

- Note that the same error occurs when using del t1[0] to delete the value of an element.

1. Tuple

1.3 Defining a tuple

- Let's learn various ways to create tuples.

Creating an empty tuple	tuple0 = tuple()	Must use empty parenthesis
Creating a tuple with a single element	tuple1 = (1,)	Must use comma
Creating a basic tuple using parentheses	tuple2 = (1, 2, 3, 4)	
Creating a simple tuple using only commas without parentheses	tuple3 = 1, 2, 3, 4	
Creating a tuple from a list	n_list = [1, 2, 3, 4] tuple4 = tuple(n_list)	

1. Tuple

1.4 Precautions when defining a tuple: integer type and tuple type variables

- When creating a tuple with only one element, if you assign it as tup = (100), it will be treated as tup = 100, and tup will become an integer, not a tuple.

```
1 tup = (100)
2 print(tup)
3 print(type(tup))
```

```
100
<class 'int'>
```

1. Tuple

1.4 Precautions when defining a tuple: integer type and tuple type variables

- Therefore, if you want to use tup as a tuple, you must insert a comma, such as tup = (100,).

```
1 tup = (100,)
2 print(tup)
3 print(type(tup))
```

```
(100,)
<class 'tuple'>
```

1. Tuple

1.4 Precautions when defining a tuple: integer type and tuple type variables

- Therefore, if you want to use tup as a tuple, you must insert a comma, such as tup = (100,).

```
1 tup = (100,)
2 print(tup)
3 print(type(tup))
```

```
(100,)
<class 'tuple'>
```

Line 1
- If tup = (100,) is used, tup becomes a tuple type.

1. Tuple

1.4 Precautions when defining a tuple: integer type and tuple type variables

- Therefore, if you want to use tup as a tuple, you must insert a comma, such as tup = (100,).

```
1 tup = (100,)
2 print(tup)
3 print(type(tup))
```

```
(100,)
<class 'tuple'>
```

Line 2-3
- The data type of tup becomes tuple.

1. Tuple

1.5 Packing, unpacking

- Packing: Putting multiple values into a single variable.
- Unpacking: If there is a packed variable, it means extracting multiple values from it.

Packing

```
1 a = (1, 2)
2 a[0]
```

```
1
```

```
1 a[1]
```

```
2
```

Unpacking

```
1 c = (3, 4)
2 x, y = c
3 x
```

```
3
```

```
1 y
```

```
4
```

1. Tuple

1.6 Tuple sorting methods

- Since tuples are immutable, the sort method cannot be used to change their elements.
- Therefore, when sorting the elements of a tuple, you can convert the tuple to a list and then use the sort method to sort the numbers.

```
1 tup = (1, 2, 5, 4, 3, 2, 9, 3, 7, 3, 9)
2 temp = list(tup)
3 temp.sort()
4 print(temp)
```

```
[1, 2, 2, 3, 3, 3, 4, 5, 7, 9, 9]
```

2. Sequence types

2.1 range function and sequence types

```
1 even_list = list(range(2, 11, 2))
2 print('even_list =', even_list)
```

```
even_list = [2, 4, 6, 8, 10]
```

- We created a list called 'even_list' that contains even numbers from 1 to 10 (including 10), and then printed the list using the print function.
- We converted the sequence generated by the range function into a list using the list function.
- Since the last argument of the range function is 2, it increments by 2 from 2 to 10. Therefore, it contains the elements 2, 4, 6, 8, 10.

2. Sequence types

2.1 range function and sequence types

- By converting range into a list, you can easily create lists of various numbers.

```
1 list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
1 list(range(2, 10))
```

```
[2, 3, 4, 5, 6, 7, 8, 9]
```

```
1 list(range(2, 10, 3))
```

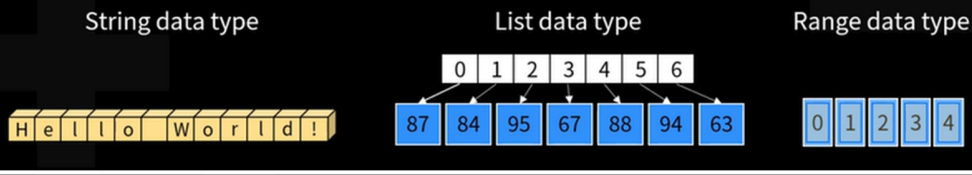
```
[2, 5, 8]
```

2. Sequence types

2.2 What is a sequence data type?

- If we examine the lists, tuples, ranges, and strings we have used so far, they have something in common. These data structures store multiple values in a consecutive(sequence) manner.
- In Python, data type that have values arranged in a consecutive manner, such as lists, tuples, ranges, and strings, is called sequence type.
- The each value contained in the sequence type is called an element.

Examples of sequence types



2. Sequence types

2.3 Index and indexing

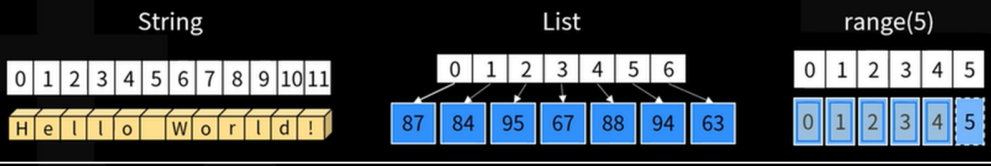
- Let's learn about index commonly used in sequence types.

Index

- It refers to the number that points to the value of an element in a list or sequence.
- The index of a list with n elements increases from 0 to n-1.

- The index of a sequence type always starts from 0.

Sequence types can reference elements using index



2. Sequence types

2.3 Index and indexing

Referencing element values through list indexing

```
1 list1 = [11, 11, 11, 22, 33, 44]
```

```
1 list1[0]
```

```
11
```

```
1 list1[2]
```

```
11
```

Referencing element values through string indexing

```
1 str1 = 'hello world'
```

```
1 str1[2]
```

```
'l'
```

```
1 str1[7]
```

```
'o'
```

2. Sequence types

2.3 Index and indexing

Referencing element values through tuple indexing

```
1 tup1 = (1, 1, 1, 2, 3, 3, 4)
```

```
1 tup1[3]
```

```
2
```

Referencing element values through range indexing

```
1 ran = range(0, 5, 1)
```

```
1 ran[4]
```

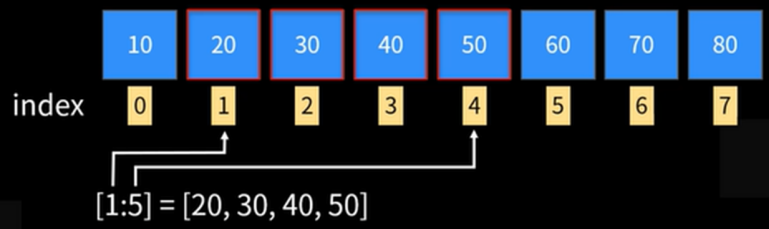
```
4
```

e.g. range types can specify the skipping interval using a step value of 1, such as range(0, 5, 1).

2. Sequence types

2.4 Slicing

- Generally, slicing of sequence types specifies the start index and the stop index.
- All sequence types use the same slicing method, following the rules below.



  - [1:5] specifies the range from index 1 to index 5-1.
  - Therefore, retrieves the elements [20, 30, 40, 50].

2. Sequence types

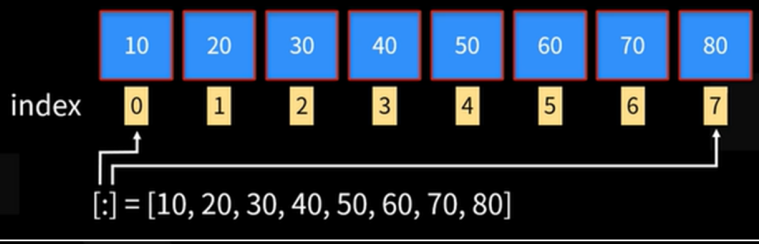
2.4 Slicing

- When retrieving the entire sequence, you can omit the stop index.
- You can slice from the second element to the last element using [1:] and similar slicing methods.

2. Sequence types

2.4 Slicing

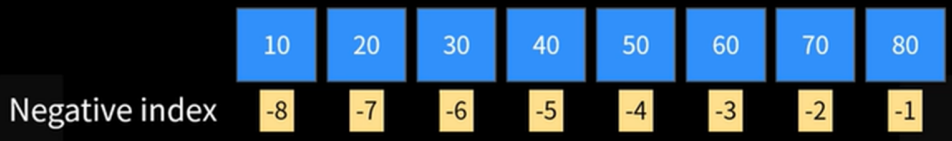
- Both the start index and the stop index can be omitted.
- If used as [:], it returns all elements.



2. Sequence types

2.5 Negative indexing

- Negative indexing is possible for sequence data types.
- The rule for negative indexing is that the index of the last element is -1, and the preceding elements are assigned -2, -3, and so on.



2. Sequence types

2.5 Negative indexing

- The string data type follows the same rule for negative indexing.

```
1 str1 = 'abcdef'
2 print(str1[-1])
f

1 print(str1[-6])
a
```

Elements of the string data type	'a'	'b'	'c'	'd'	'e'	'f'
Negative index	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

2. Sequence types

2.5 Negative indexing

- The tuple data type follows the same rule for negative indexing.

```
1  tup1 = (11, 22, 33, 44, 55, 66, 77)
2  print(tup1[-1])
```

```
77
```

```
1  print(tup1[-6])
```

```
22
```

Elements of the tuple data type	11	22	33	44	55	66	77
Negative index	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

2. Sequence types

2.5 Negative indexing

- The range data type follows the same rule for negative indexing.

```
1  ran = range(1, 7)
2  print(ran[-6])
```

```
1
```

```
1  print(ran[-1])
```

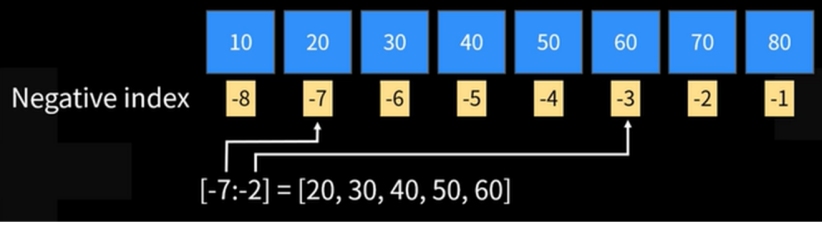
```
6
```

Elements of the range data type	1	2	3	4	5	6
Negative index	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

2. Sequence types

2.6 Slicing using negative indexing

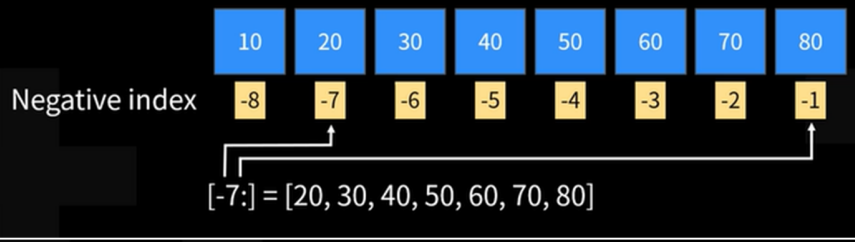
- Let's learn about slicing using negative indexing.
- When using the range of negative indexes as [-7:-2], the [20, 30, 40, 50, 60] elements are included.



2. Sequence types

2.6 Slicing using negative indexing

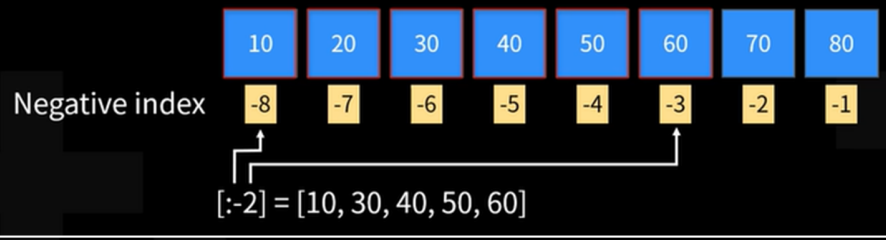
- When using negative indexing, if the stop index is omitted, it returns up to the last element of the list.



2. Sequence types

2.6 Slicing using negative indexing

- In the case of slicing using negative indexing, the start index can also be omitted.
- [:-2] returns the values up to the element with index -3.



2. Sequence types

2.7 Membership operator for specific element search: in, not in

- The membership operator 'in', 'not in' is operator that return True or False.
- They are used to check if a specific element is present inside data structures such as strings, lists, and tuples.

- 'in' operator: Returns True if the member is present, and False if it is not.
- 'not in' operator: Returns False if the member is present, and True if it is not.

2. Sequence types

2.7 Membership operator for specific element search: in, not in

- In the following example, since 10 is in list1, '10 in list1' returns True, and '10 not in list1' returns False.

```
1 list1 = [10, 20, 30, 40]
2 10 in list1
```

```
True
```

```
1 list1 = [10, 20, 30, 40]
2 10 not in list1
```

```
False
```

2. Sequence types

2.7 Membership operator for specific element search: in, not in

- The in operator can be used to search for a specific value within a tuple.

```
1 tup = (1, 2, 3, 4)
2 3 in tup
```

```
True
```

2. Sequence types

2.7 Membership operator for specific element search: in, not in

- The in operator can also be used with range sequences generated by the range function.

```
1 11 in range(10)
```

```
False
```

2. Sequence types

2.7 Membership operator for specific element search: in, not in

- It can be used with strings to search for a specific value.

```
1 'a' in 'abcd'
```

Since the alphabet 'a' is included in 'abcd', it evaluates to True.

```
True
```

2. Sequence types

2.8 len function

- By subtracting 1 from the length obtained using the len function, we can access the last index.
- Let's create a list called nations with the elements 'Korea', 'China', 'Russia', 'Malaysia', and use the index len(nations)-1 to print the last element of the list.

```
1 nations = ['Korea', 'China', 'Russia', 'Malaysia']
2 print('Last element of nations :', nations[len(nations)-1])
```

```
Last element of nations : Malaysia
```

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Let's explore the count method, which counts the number of specific elements in a sequence.

The count method returns the number of occurrences of elements within a sequence data type.

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Let's compare the results of len and count for a range data type called ran.

```
1 ran = range(0, 5, 1)
2 print(len(ran))
3 print(ran.count(2))
```

```
5
1
```

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Let's compare the results of len and count for a range data type called ran.

```
1 ran = range(0, 5, 1)
2 print(len(ran))
3 print(ran.count(2))
```

```
5
1
```

Line 2

- ran contains elements 0, 1, 2, 3, 4, so the count is 5.
- When we use the len function to count the elements, it returns 5.

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Let's compare the results of len and count for a range data type called ran.

```
1 ran = range(0, 5, 1)
2 print(len(ran))
3 print(ran.count(2))
```

```
5
1
```

Line 3
- On the other hand, let's use the count method to find out how many times 2 appears in the ran variable.
- Since 2 appears only once, it returns 1.

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Output of the count method for a list.

```
1 list1 = [11, 11, 11, 22, 33, 44]
2 print(list1.count(11))
```

```
3
```

- Since there are 3 occurrences of the element 11 in list1, it returns 3.

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Output of the count method for a tuple.

```
1 tup1 = (11, 11, 11, 22, 33, 44)
2 print(tup1.count(11))
```

```
3
```

- tup1 contains 3 occurrences of 11.

2. Sequence types

2.9 Count method for determining the number of specific elements in a sequence

- Output of the count method for a string.

```
1 str1 = 'hello world'
2 print(str1.count('l'))
```

```
3
```

- str1 contains 3 occurrences of the character 'l'.

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- Sequence data types can be concatenated using the concatenation operator (+).

> ! However, the range data type cannot be concatenated using the + operator.

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- Let's learn how to concatenate lists.

```
1 list1 = [11, 22, 33, 44]
2 list2 = [55, 66]
3 print(list1)
4 print(list1 + list2)
```

The + operator can be used.

```
[11, 22, 33, 44]
[11, 22, 33, 44, 55, 66]
```

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- Let's learn how to concatenate tuples.

```
1 tup1 = (1, 2, 3)
2 tup2 = (4, 5, 6)
3 print(tup1 + tup2)
```

The + operator can be used.

```
(1, 2, 3, 4, 5, 6)
```

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- Let's learn how to concatenate strings.

```
1 str1 = 'hello '
2 str2 = 'world'
3 print(str1 + str2)
```

The + operator can be used.

```
hello world
```

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- Let's explore concatenation with the range data type.
- An error occurs when using the + operator with range.

TypeError

```
1 range(10) + range(10, 20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[45], line 1
----> 1 range(10) + range(10, 20)

TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

Among the sequence data types, range cannot be concatenated with the + operator.

3. Precautions when using sequence data types

3.1 Concatenation operator for sequence data types

- In such cases, range can be converted to a list or tuple and then concatenated.

```
1 list(range(10)) + list(range(10, 20))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
1 tuple(range(10)) + tuple(range(10, 20))
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Sequence data types can also use the repetition operator, which repeats the elements of the data type.
  - The syntax is 'sequence type * integer'.
  - However, range cannot use the * operator.

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Let's learn how to repeat elements in a list using the * operator.

```
1 list1 = [11, 22, 33, 44] * 2
2 print(list1)
```

```
[11, 22, 33, 44, 11, 22, 33, 44]
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Let's learn how to repeat elements in a tuple using the * operator.

```
1 tup1 = (1, 2, 3)
2 print(tup1 * 2)
```

```
(1, 2, 3, 1, 2, 3)
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Let's learn how to repeat characters in a string using the * operator.

```
1 str2 = 'hello'
2 print(str2 * 3)
```

```
hellohellohello
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Let's learn about duplicating methods for range. If you use the * operator on a range type, an error will occur.

TypeError

```
1 range(10) * 3
```

The range data type cannot be used with the + operator.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 range(10) * 3

TypeError: unsupported operand type(s) for *: 'range' and 'int'
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Let's learn about duplicating methods for range. If you use the * operator on a range type, an error will occur.

TypeError

```
1 range(10) * 3
```

Similarly, the range data type cannot be used with the * operator.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 range(10) * 3

TypeError: unsupported operand type(s) for *: 'range' and 'int'
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- Therefore, you can convert it to a list data type or tuple data type to duplicate it as follows.

```
1 ran = list(range(5)) * 3
2 print(ran)
```

```
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

```
1 ran = tuple(range(5)) * 3
2 print(ran)
```

```
(0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4)
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- The following is an example of incorrect usage of the duplication operator.
- If you use the duplication operator on two lists, an error will occur.

TypeError

```
1 list1 = [11, 22, 33, 44]
2 list2 = [55, 66]
3 print(list1 * list2)
```

Duplication operators cannot be used between lists, tuples, or string data types.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[54], line 3
      1 list1 = [11, 22, 33, 44]
      2 list2 = [55, 66]
----> 3 print(list1 * list2)

TypeError: can't multiply sequence by non-int of type 'list'
```

3. Precautions when using sequence data types

3.2 Operator that duplicate sequence types

- The following is an example of incorrect usage of the duplication operator.
- If you use the duplication operator on two lists, an error will occur.

TypeError

```
1 list1 = [11, 22, 33, 44]
2 list2 = [55, 66]
3 print(list1 * list2)
```

When performing duplication operations on any sequence data type, you must follow the syntax; sequence data type * integer.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[54], line 3
      1 list1 = [11, 22, 33, 44]
      2 list2 = [55, 66]
----> 3 print(list1 * list2)

TypeError: can't multiply sequence by non-int of type 'list'
```

3. Precautions when using sequence data types

3.3 Comparison operator

- Comparison of “greater than” or “less than” for sequence data types is determined by comparing the elements of the two sequence data types.

```
1 a = ('A', 'B', 'C')
2 b = ('A', 'B', 'D')
```

```
1 a > b
```

```
False
```

```
1 a < b
```

```
True
```

3. Precautions when using sequence data types

3.3 Comparison operator

```
1 ord('C')
```

```
67
```

```
1 ord('D')
```

```
68
```

The code value of ‘C’ in tuple a is 67, and ‘D’ in tuple b is 68, so a > b is False and a < b is True.

- The ord function is a built-in function in Python that converts a character to its corresponding Unicode value.
- In other words, it returns the Unicode code point value of the given character.

3. Precautions when using sequence data types

3.3 Comparison operator

- Comparison examples for lists.

```
1 a = ['A', 'B', 'C']
2 b = ['A', 'B', 'D']
```

```
1 a > b
```

```
False
```

```
1 a < b
```

```
True
```

3. Precautions when using sequence data types

3.3 Comparison operator

- Comparison examples for strings.

```
1 a = 'ABC'
2 b = 'ABD'
```

```
1 a > b
```

```
False
```

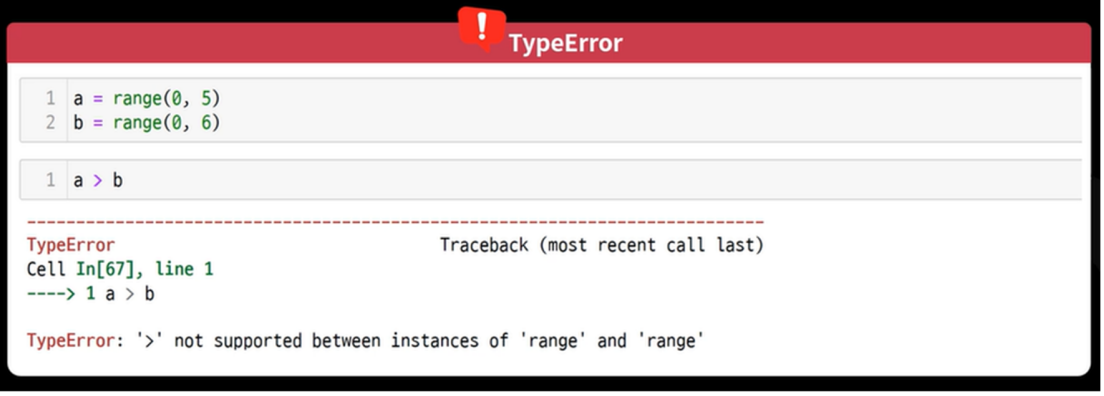
```
1 a < b
```

```
True
```

3. Precautions when using sequence data types

3.3 Comparison operator

- An error occurs when performing comparison operations on range types.



3. Precautions when using sequence data types

3.3 Comparison operator

- The range data type cannot be used with comparison operators.

TypeError

```
1 a < b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[68], line 1
----> 1 a < b

TypeError: '<' not supported between instances of 'range' and 'range'
```

4. Immutable data types

4.1 What are immutable data types?

- Immutable data types refer to data types that cannot be changed after they are created.
- In other words, they cannot be modified internally. They can only be reassigned.
- Tuples are a representative example. Tuples cannot modify their internal elements. A new value must be assigned.

4. Immutable data types

4.2 Immutable data types and the is operator

- This code determines whether two variables with the same value are the same object using the is operator.

> We learned that the is operator returns True if the two objects on both sides are the same, and False otherwise.

- Let's compare the results of code that checks two variables with the same elements in a list using the is operator and code that checks two variables with the same elements in a string using the is operator.

4. Immutable data types

4.2 Immutable data types and the is operator

- First, the results of the code that checks two variables with the same elements in a list using the is operator are as follows.

```
list1 = [10, 20, 30]
list2 = [10, 20, 30]
if list1 == list2:
    print("It's same.")
    if list1 is list2:
        print("It's same object.")
    else:
        print("It's different object.")
else:
    print("It's different.")
```

```
It's same.
It's different object.
```

4. Immutable data types

4.2 Immutable data types and the is operator

- The results of the code that checks two variables with the same elements in a string using the is operator are as follows.

```
string1 = 'ABC'
string2 = 'ABC'
if string1 == string2:
    print("It's same.")
    if string1 is string2:
        print("It's same object.")
    else:
        print("It's different object.")
else:
    print("It's different.")
```

```
It's same.
It's same object.
```

4. Immutable data types

4.2 Immutable data types and the is operator

- The results of the code that checks two variables with the same elements in a stri
using the is operator are as follows.

Let's compare it with the results of the code that checks two variables with the same elements in a list using the is operator that we saw earlier.

4. Immutable data types

4.2 Immutable data types and the is operator

- The results of the code that checks two variables with the same elements in a string using the is operator are as follows.

```
1  string1 = 'ABC'
2  string2 = 'ABC'
3  if string1 == string2:
4      print("It's same.")
5      if string1 is string2:
6          print("It's same object.")
7      else:
8          print("It's different object.")
9  else:
10     print("It's different.")
```

```
It's same.
It's same object.
```

string1 and string2 reference the 'ABC' string. In this case, unlike lists, string1 and string2 reference the same object. Why is that?

It is because strings are immutable data types. Let's take a closer look at this.

4. Immutable data types

4.3 Immutable data types and the id function

- Let's check the id values of each variable.

```
1  print(id(list1), 'vs.', id(list2))
2  print(id(string1), 'vs.', id(string2))
```

```
4482160384 vs. 4482167104
4428333048 vs. 4428333048
```

4. Immutable data types

4.3 Immutable data types and the id function

- Let's check the id values of each variable.

```
1  print(id(list1), 'vs.', id(list2))
2  print(id(string1), 'vs.', id(string2))
```

```
4482160384 vs. 4482167104
4428333048 vs. 4428333048
```

Line 1
- list1 and list2 have different id values.

4. Immutable data types

4.3 Immutable data types and the id function

- Let's check the id values of each variable.

```
1 print(id(list1), 'vs.', id(list2))
2 print(id(string1), 'vs.', id(string2))
```

```
4482160384 vs. 4482167104
4428333048 vs. 4428333048
```

Line 2
- However, string1 and string2 have the same id value.

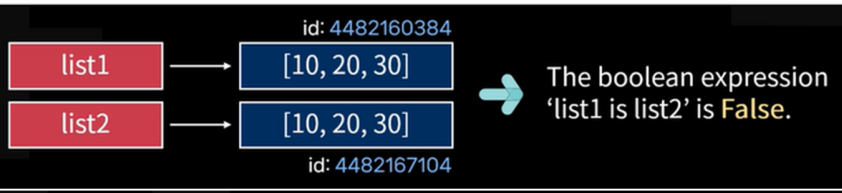
4. Immutable data types

4.3 Immutable data types and the id function

- Since the list [10, 20, 30] is stored in different memory locations, the id values of list1 and list2 are different.
- The 'is' operator compares id values, so it returns False.

```
1 print(id(list1), 'vs.', id(list2))
2 print(id(string1), 'vs.', id(string2))
```

```
4482160384 vs. 4482167104
4428333048 vs. 4428333048
```



4. Immutable data types

4.3 Immutable data types and the id function

- For the 'str' data type, a table is used to store its location, and if the same string is assigned, it refers to the same location.
- This is one of Python's optimization features, which reduces memory usage and improves performance for small-sized repeated strings.

4. Immutable data types

4.3 Immutable data types and the id function

- Therefore, since the string 'ABC' has the same storage location, the id values of string1 and string2 are the same.
- The 'is' operator compares id values, so it returns True.

```
1 print(id(list1), 'vs.', id(list2))
2 print(id(string1), 'vs.', id(string2))
```

```
4482160384 vs. 4482167104
4428333048 vs. 4428333048
```

4. Immutable data types

4.4 Immutable data types and the assignment operator

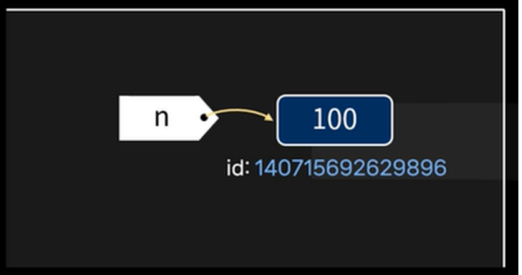
- The variable name we use is essentially a reference to an object.
- In the following code, the object with a value of 100 is accessed through a variable named 'n'.

```
1 n = 100
2 id(100)
```

```
140715692629896
```

```
1 id(n)
```

```
140715692629896
```



4. Immutable data types

4.4 Immutable data types and the assignment operator

- When there is a numeric type with a value of 100 and it is referenced by 'n', it is possible to access 100 through the variable 'n'.

```
1 n = 100
2 m = n
3 id(n)
```

```
140715692629896
```

```
1 id(m)
```

```
140715692629896
```

4. Immutable data types

4.4 Immutable data types and the assignment operator

- When there is a numeric type with a value of 100 and it is referenced by 'n', it is possible to access 100 through the variable 'n'.

```
n = 100
m = n
id(n)
```

```
140715692629896
```

Another variable 'm' can also be used to access the value of 100 using the assignment operator.

```
id(m)
```

```
140715692629896
```

4. Immutable data types

4.4 Immutable data types and the assignment operator

- When there is a numeric type with a value of 100 and it is referenced by 'n', it is possible to access 100 through the variable 'n'.

```
n = 100
m = n
id(n)
```

```
140715692629896
```

The assignment operator '=' performs referencing and re-referencing for the value of 100.

```
id(m)
```

```
140715692629896
```

4. Immutable data types

4.4 Immutable data types and the assignment operator

- When there is a numeric type with a value of 100 and it is referenced by 'n', it is possible to access 100 through the variable 'n'.

```
n = 100
m = n
id(n)
```

```
140715692629896
```

```
id(m)
```

```
140715692629896
```

The two variables refer to the same object.

4. Immutable data types

4.4 Immutable data types and the assignment operator

- Integer, float, string, boolean, and tuple are immutable data types.
- Therefore, if their values are the same, they refer to the same storage location.
- This is done to efficiently use memory.

4. Immutable data types

4.4 Immutable data types and the assignment operator

```
1 n = 100
2 m = 100
3 print(id(n))
4 print(id(m))
```

```
140715692629896
140715692629896
```

4. Immutable data types

4.4 Immutable data types and the assignment operator

```
1 n = 100
2 m = 100
3 print(id(n))
4 print(id(m))
```

```
140715692629896
140715692629896
```

Line 3-4
- Both m and n refer to the number 100, so the id of the two objects is the same.

4. Immutable data types

4.4 Immutable data types and the assignment operator

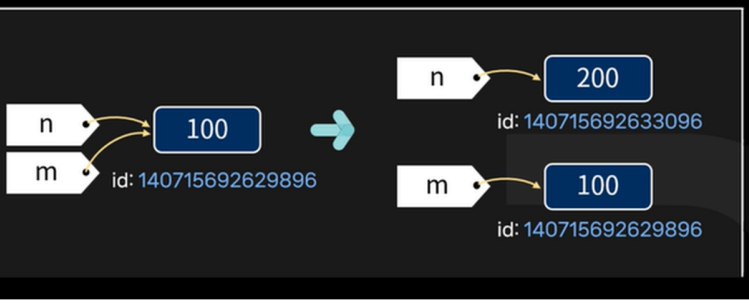
- When a different value is assigned, the storage location changes, so it becomes a different object.

```
1 n = 100
2 m = n
3 n = 200
4 id(n)
```

```
140715692633096
```

```
1 id(m)
```

```
140715692629896
```



4. Immutable data types

4.4 Immutable data types and the assignment operator

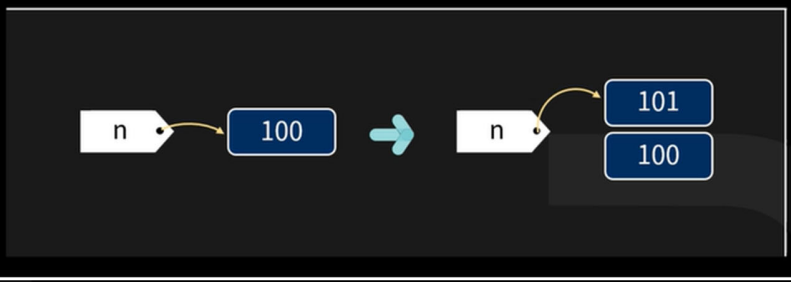
- Also, by assigning n = n + 1, the newly assigned n and the previous n refer to different objects.

```
1 n = 100
2 id(n)
```

```
140715692629896
```

```
1 n = n + 1
2 id(n)
```

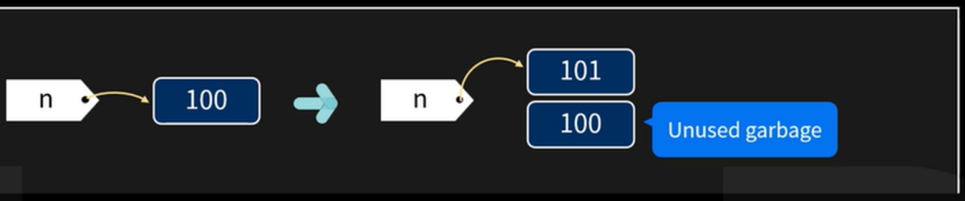
```
140715692629928
```



4. Immutable data types

4.4 Immutable data types and the assignment operator

- At this point, the value '100' becomes garbage, which is no longer used.



- Garbage refers to objects that exist in memory but no longer have any references to access them.
- Since there are no variables referencing this object anymore, it causes memory waste.

4. Immutable data types

4.4 Immutable data types and the assignment operator

- At this point, the value '100' becomes garbage, which is no longer used.

Therefore, periodic garbage memory cleanup is necessary, and this memory management procedure is called garbage collection.