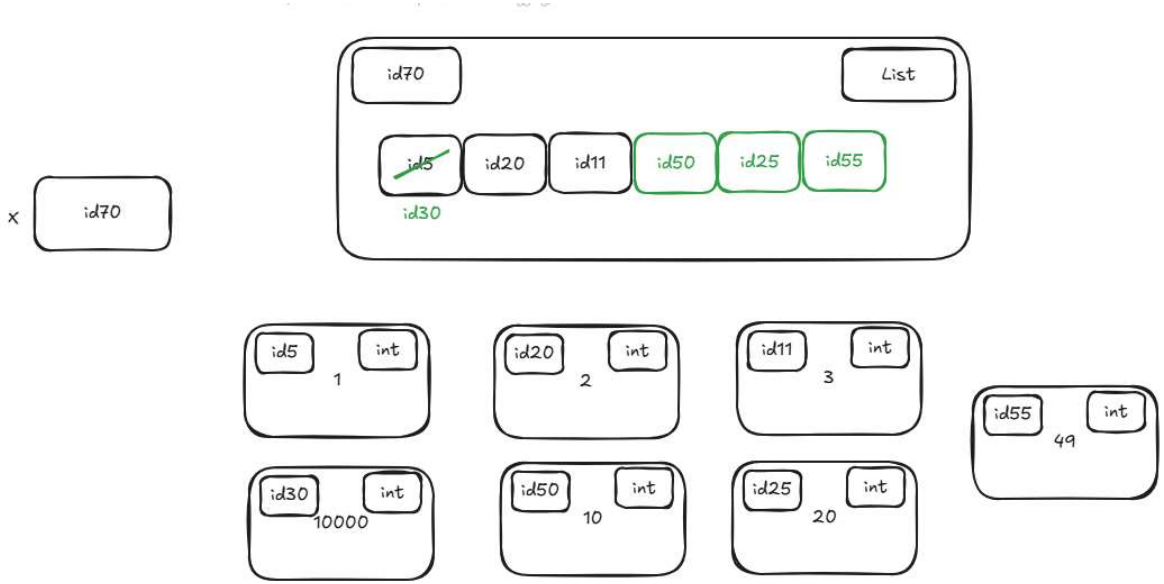
Memory Model (List):

```
>>> x = [1,2,3]
>>> type(x)
<class 'list'>
>>> id(x)
2629899132416
>>>
>>> x[0] = 10000
>>> x
[10000, 2, 3]
>>> id(x)
2629899132416
>>> x.extend([10, 20, 49])
>>> x
[10000, 2, 3, 10, 20, 49]
>>> id(x)
2629899132416
```



- **Tuple**:

- A Tuple is a container used to hold a series of comma-separated values between parenthesis (), such as (x,y) coordinates.
- Tuples are like list, except they are immutable.
- A Tuple is a collection which is ordered and unchangeable.
- The parenthesis is optional, but it is a good practice to use.
- An empty tuple can be created.

```
T = 2,3,4,5
print(T)
(2, 3, 4, 5)
print(type(T))
<class 'tuple'>

T = (2,3,4,5)
print(T)
(2, 3, 4, 5)
print(type(T))
<class 'tuple'>

T = ("SMIT")
print(type(T))
<class 'str'>
T = ("SMIT",)
print(type(T))
<class 'tuple'>
```

- **Tuple Properties:**

A tuple has following properties:

1. Positive and negative indexes.
2. Slicing.
3. Updating.

```
T = ("apple", "cherry", "banana")
L = list(T)
L
['apple', 'cherry', 'banana']
print(type(L))
<class 'list'>

Tuple = tuple(L)
Tuple
('apple', 'cherry', 'banana')
print(type(Tuple))
<class 'tuple'>
```

- **Tuple (Unpacking):**

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
apple
print(yellow)
banana
print(red)
cherry
```

- **Tuple (Using Asterisk):**

```
fruits = ("apple", "banana", "cherry", "apricot", "mango")
(green, yellow, *red) = fruits
print(green)
apple
print(yellow)
banana
print(red)
['cherry', 'apricot', 'mango']

# Try this statement  (green, *yellow, red) = fruits

(*green, *yellow, red) = fruits
SyntaxError: multiple starred expressions in assignment
```

- **Tuple (Built-in Functions):**

➢ count() → Returns the number of times a specified value occurs in a tuple.

➢ index() → Searches the tuple for a specified value and returns the position of where it was found.

```
fruits
('apple', 'banana', 'cherry', 'apricot', 'mango')
fruits.count("apple")
1
fruits.index("apricot")
3
```

```
# Function example
def my_func(*args):
    print(type(args))
    for arg in args:
        print(arg)
```

```
my_func(1,2,3,4,5)
<class 'tuple'>
1
2
3
4
5
```

- **Dictionary:**

Dictionary in Python is a collection of keys and values, used to store data like a map, unlike other data types which hold only a single value as an element. The dictionary is defined into elements Keys and Values.

- Accessed by keys, not offset position.
- Variable-length, heterogeneous and arbitrarily nestable.
- Mutable mapping.

- **Dictionary (get, keys)**

```
dict
<class 'dict'>
D = {"1": "corolla", "2": "farrari", "3": "alto", "4": "cultus", "5": "1993"}
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'cultus', '5': '1993'}
x = D["3"]
print(x)
alto
y = D.get("4")
y
'cultus'
dict_keys = D.keys()
dict_keys
dict_keys(['1', '2', '3', '4', '5'])
# insert element into dictionary
D["6"] = "Alto VXL"
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'cultus', '5': '1993', '6'
: 'Alto VXL'}
dict_keys
dict_keys(['1', '2', '3', '4', '5', '6'])
```

```
# Example of get() method in Dictionary
# If key is not present then get() does not throw an error
# and return the None value because default is set to None
words = ['apple', 'cherry', 'apple', 'cherry','raspberry']

freq = {}
for word in words:
    freq[word] = freq.get(word, 0) + 1


print(freq)
{'apple': 2, 'cherry': 2, 'raspberry': 1}
```

- **Dictionary (values, items):**

```
dict_values = D.values()
dict_values
dict_values(['corolla', 'farrari', 'alto', 'cultus', '1993', 'Alto VXL'])
D["year"] = 2022
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'cultus', '5': '1993', '6'
: 'Alto VXL', 'year': 2022}
dict_values
dict_values(['corolla', 'farrari', 'alto', 'cultus', '1993', 'Alto VXL', 2022
])

dict_items = D.items()
dict_items
dict_items([('1', 'corolla'), ('2', 'farrari'), ('3', 'alto'), ('4', 'cultus'
), ('5', '1993'), ('6', 'Alto VXL'), ('year', 2022)])
D["year"] = 1947
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'cultus', '5': '1993', '6'
: 'Alto VXL', 'year': 1947}
dict_items
dict_items([('1', 'corolla'), ('2', 'farrari'), ('3', 'alto'), ('4', 'cultus'
), ('5', '1993'), ('6', 'Alto VXL'), ('year', 1947)])
```

- **Dictionary (update, pop, popitem)**

```
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'cultus', '5': '1993', '6'
: 'Alto VXL', 'year': 1947}
D["4"] = "apple"
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'apple', '5': '1993', '6':
'Alto VXL', 'year': 1947}
D.update({"4": "altis"})
D
{'1': 'corolla', '2': 'farrari', '3': 'alto', '4': 'altis', '5': '1993', '6':
'Alto VXL', 'year': 1947}
D.update({"4": "cultus", "3": "honda"})
D
{'1': 'corolla', '2': 'farrari', '3': 'honda', '4': 'cultus', '5': '1993', '6
': 'Alto VXL', 'year': 1947}
# dictionary pop
D.pop("3")
'honda'
D
{'1': 'corolla', '2': 'farrari', '4': 'cultus', '5': '1993', '6': 'Alto VXL',
'year': 1947}
D.popitem()
('year', 1947)
D
{'1': 'corolla', '2': 'farrari', '4': 'cultus', '5': '1993', '6': 'Alto VXL'}
```

- **Dictionary (copy, del, clear):**

```
D_copy = D.copy()
D_copy
{'1': 'corolla', '2': 'farrari', '4': 'cultus', '5': '1993', '6': 'Alto VXL'}
del D['2']
D
{'1': 'corolla', '4': 'cultus', '5': '1993', '6': 'Alto VXL'}
del D
D
Traceback (most recent call last):
  File "<pyshell#621>", line 1, in <module>
    D
NameError: name 'D' is not defined

D_copy.clear()
D_copy
{}
```

- **Dictionary (Built-in functions) [Summary]**

- clear() → Remove all items from the dictionary.
- copy() → Returns a copy of the dictionary.
- get() → Returns the value of the specified key. If key not present then return None (default).
- items() → Returns a list containing a tuple of each key value pair.
- keys() → Returns a list containing dictionary's keys.
- fromkeys() → Returns a dictionary with specified keys and values.
- pop() → Remove the element with specified key.
- popitem() → Removes the last inserted key-value pair.
- update() → Update dictionary with specified key-value pair.
- values() → Returns a list containing dictionary's values.
- setdefault() → Returns the value of the specified key. If the key does not exist: insert the key with specified value.
- del → keyword used to remove the item with the specified key name

Nested Dictionary:

A nested dictionary is a dictionary where the value associated with a key is itself an another dictionary.

```
# creating a nested dictionary for student details
students = {
    'student1': {'name': 'Alice', 'age': 20, 'grade': 'A'},
    'student2': {'name': 'Bob', 'age': 22, 'grade': 'B'},
    'student3': {'name': 'Charlie', 'age': 21, 'grade': 'A+'}
}
print(students)
{'student1': {'name': 'Alice', 'age': 20, 'grade': 'A'}, 'student2': {'name':
'Bob', 'age': 22, 'grade': 'B'}, 'student3': {'name': 'Charlie', 'age': 21, '
grade': 'A+'}}

# access detail of student2
students["student2"]
{'name': 'Bob', 'age': 22, 'grade': 'B'}
# access name of student 2
students["student2"]["name"]
'Bob'
# modify grade of student2
students["student2"]["grade"] = "A-"
print(students)
{'student1': {'name': 'Alice', 'age': 20, 'grade': 'A'}, 'student2': {'name':
'Bob', 'age': 22, 'grade': 'A-'}, 'student3': {'name': 'Charlie', 'age': 21,
'grade': 'A+'}}
# Adding a new student
students['student4'] = {'name': 'David', 'age': 19, 'grade': 'B+'}
print(students['student4'])
SyntaxError: multiple statements found while compiling a single statement
# Adding a new student
students['student4'] = {'name': 'David', 'age': 19, 'grade': 'B+'}
print(students["student4"])
{'name': 'David', 'age': 19, 'grade': 'B+'}
```

Tasks on Dictionary:

1. Create the dictionary:

   {"CP": "Computer Programming", "FCE": "Fundamental of Computer Engineering", "SMIT": "Saylani Mass Information Technology"}

2. Write a statement to add “AI”: “Artificial Intelligence” in the dictionary of Q1.
3. Write program to find out whether the given key (Chinese) is the part of dictionary in Q1.
4. Write a program that takes a list of multiple choice responses for e.g. [‘a’, ‘b’, ‘c’] and prints a dictionary of question response pairs

   {“Q1”: “a”, “Q2”: “b”, “Q3”: “c”}

```
# Dictionary iterating over items
Dict = {'1': 'Saylani', '2': 'Mass', '3': 'IT'}
print(Dict.items())
dict_items([('1', 'Saylani'), ('2', 'Mass'), ('3', 'IT')])
for key, value in Dict.items():
    print(f"{key}: {value}")


1: Saylani
2: Mass
3: IT
```

```
def my_func(**kwargs):
    print(type(kwargs))
    for kwarg in kwargs.items():
        print(kwarg)


my_func(first='computer', second='programming')
<class 'dict'>
('first', 'computer')
('second', 'programming')
```

- **Set:**

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable (immutable), unindexed and does not allow duplicate values.
- Unordered means that the items in a set do not have a defined order.
- Unchangeable means that we can add or remove items from a set after its creation. However, the individual elements themselves are generally immutable. You cannot directly modify an element within a set.
- Sets cannot have two items with the same values.
- Sets are typically created by enclosing a comma-separated sequence of items within a curly braces {}.
- Set supports operations inspired by mathematical set theory.

Set

```
# set creation
Set = {1,2,3,"apple", "raspberry", "banana"}
print(type(Set))
<class 'set'>
```

```
# convert list to set
L = [1,2,3,"apple", "mango", "raspberry"]
print(type(L))
<class 'list'>
S = set(L)
print(type(S))
<class 'set'>
print(S)
{1, 2, 'apple', 3, 'raspberry', 'mango'}
```

```
# convert tuple to set
T = (1,2,3, "apple", "mango", "raspberry")
print(type(T))
<class 'tuple'>
S = set(T)
print(type(S))
<class 'set'>
print(S)
{1, 2, 'apple', 3, 'raspberry', 'mango'}
```

```
S = {1,2,4,2,3,6,6}
S
{1, 2, 3, 4, 6}
```

```
L = [1,2,4,2,3,6,6,"apple", "apple", "orange"]
S = set(L)
print(S)
{1, 2, 3, 4, 'apple', 6, 'orange'}
```

Set (add, update):

```
# add
thisset = {"apple", "banana", "orange"}
print(thisset)
{'banana', 'apple', 'orange'}
thisset.add("mango")
print(thisset)
{'banana', 'apple', 'mango', 'orange'}

# update
x = {"apple", "banana", "cherry"}
y = {"SMIT", "Python", "apple"}
x.update(y)
print(x)
{'apple', 'SMIT', 'banana', 'cherry', 'Python'}
```

Set (difference, remove):

```
# difference
x = {"apple", "banana", "cherry"}
y = {"SMIT", "Python", "apple"}
z = x.difference(y)
print(z)
{'cherry', 'banana'}
z = y.difference(x)
print(z)
{'SMIT', 'Python'}

# remove
fruits = {"apple", "banana", "cherry"}
fruits.remove("cherry")
print(fruits)
{'apple', 'banana'}
fruits.remove("orange")
Traceback (most recent call last):
  File "<pyshell#913>", line 1, in <module>
    fruits.remove("orange")
KeyError: 'orange'
```

Set (discard, pop):

```
# discard
fruits = {"apple", "banana", "cherry"}
fruits.discard("apple")
print(fruits)
{'cherry', 'banana'}
fruits.discard("orange")
print(fruits)
{'cherry', 'banana'}

# pop
fruits = {"apple", "banana", "cherry"}
print(fruits)
{'apple', 'cherry', 'banana'}
fruits.pop()
'apple'
print(fruits)
{'cherry', 'banana'}
```

Set (union, intersection):

```
# union
x = {"apple", "banana", "cherry"}
y = {"SMIT", "Python", "apple"}
z = x.union(y)
print(z)
{'apple', 'SMIT', 'banana', 'cherry', 'Python'}

# intersection
x
{'apple', 'cherry', 'banana'}
y
{'apple', 'SMIT', 'Python'}
z = x.intersection(y)
print(z)
{'apple'}
```

Set (Built-in functions) [summary]:

- *add()* → Adds an element to the set.
- *update()* → Adds elements from another set, list or any iterable into the current set.
- *difference()* → Return elements of one set which is/are not present in other.
- *remove()* → Removes the specified element from the set. Raise error if element does not exist in the set.
- *discard()* → Removes the specified element, no error occurs if item doesn't exist.
- *pop()* → Removes the first element from the set.
- *union()* → Returns a set containing combined elements of sets.
- *intersection()* → Returns a set containing common elements between two sets.
- *symmetric_difference()* → Returns unique elements in both sets.

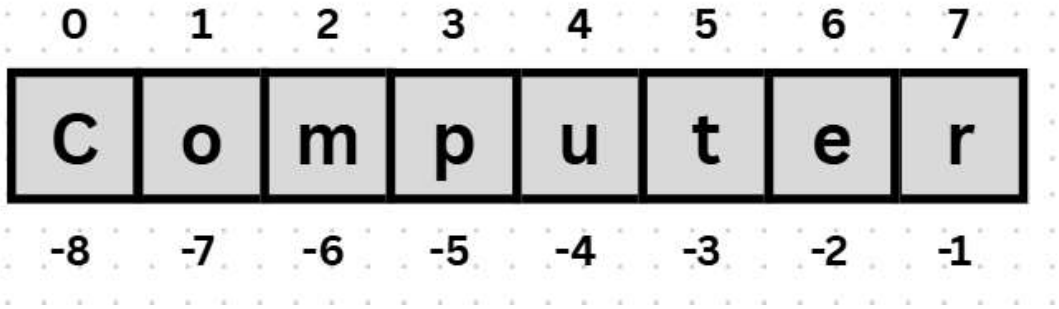
- **Strings slicing:**

Slicing means to extract specific portion (subsequences) from sequence.

Syntax → [start:end:step]

The third parameter defines:

- Difference between the indexes to be accessed.
- Direction of access i.e. negative difference defines the access direction from right to left.



```
>>> S = 'Computer'
>>> S[1:5:1]          # Same as S[1:5]
'ompu'
>>> S[1:5:2]
'op'
>>> S[1:3]          # Slice of S from offset 1 through 2 (not 3)
'om'
>>> S[1:]           # Slice of S from offset 1 to the end (1:len(S))
'omputer'
>>> S[:3]
'Com'
>>> S[5:1:-1]
'tupm'
>>> S[5:1:-2]
'tp'
>>> S[::-1]
'retupmoC'
>>> S[::1]           # Same as S[::]
'Computer'
>>> S[-4:]           # access last 4 character of string same as S[4:]
'uter'
>>> S[:-4]           # access all character except the last 4 same as S[:4]
'Comp'
```

- **String concatenation and repetition:**

Joining strings using + operator and repeating strings using * operators.

```
string1 = "Hello"
string2 = "Python"
combined_string = string1 + " " + string2
repeated_string = string1 * 3

print(combined_string)
Hello Python
print(repeated_string)
HelloHelloHello
```

- **String case-conversion:**

Methods like *lower(), upper(), capitalize(), title()* and *swapcase()* to change the case of characters.

```
string = "PyThOn"
lowercase_text = string.lower()
uppercase_text = string.upper()
capitalize_text = string.capitalize()
title_text = string.title()
swapcase_text = string.swapcase()

print(lowercase_text)
python
print(uppercase_text)
PYTHON
print(capitalize_text)
Python
print(title_text)
Python
print(swapcase_text)
pYtHoN

string = "this is a sample string for title case"
print(string.title())
This Is A Sample String For Title Case
```

- **String searching and replacing:**

Methods like *find(), index(), replace()* is used to locate and modify substrings

```
sentence = "Python is powerful."
position = sentence.find("powerful")
print(position)
10
new_string = sentence.replace("powerful", "amazing")
print(new_string)
Python is amazing.
print(sentence)
Python is powerful.
```

```
sentence = "Python is powerful."
print(sentence.find("are"))
-1
print(sentence.index("are"))
Traceback (most recent call last):
  File "<pyshell#371>", line 1, in <module>
    print(sentence.index("are"))
ValueError: substring not found
```

```
sentence = "Hello world, Hello Python"
# replace all occurences
new_string = sentence.replace("Hello", "Hi")
print(new_string)
Hi world, Hi Python
# replace a specific number of occurences
new_string = sentence.replace("Hello", "Hi", 1)
print(new_string)
Hi world, Hello Python
```

- **String splitting and joining:**

The *split()* method is used to break a string into a list of substring based on a delimiter, and *join()* method to concatenate elements of an iterable into a single string.

```
data = "apple,banana,orange"
fruits = data.split(",")
print(fruits)
['apple', 'banana', 'orange']
joined_string = "-".join(fruits)
print(joined_string)
apple-banana-orange
```

- **String stripping:**

Methods like *strip(), lstrip(), rstrip()* is used to remove leading/trailing whitespaces or specified character.

```
padded_string = " hello "
len(padded_string)
7
trimmed_string = padded_string.strip()
print(trimmed_string)
hello
len(trimmed_string)
5
trimmed_string_left = padded_string.lstrip()
print(trimmed_string_left)
hello
len(trimmed_string_left)
6
trimmed_string_right = padded_string.rstrip()
print(trimmed_string_right)
 hello
len(trimmed_string_right)
6
```

```
specific_character_text = "---Python---"
stripped_specific = specific_character_text.strip("-")
print(stripped_specific)
Python
```

- **String count:**

The *count()* method is used to determine the number occurrences of a specified substring within a given string.

```
# counting occurence of a string character
text = "hello world"
count_o = text.count("o")
print(count_o)
2

# counting occurence of a substring
sentence = "the quick brown fox jumps over the lazy dog"
count_the = sentence.count("the")
print(count_the)
2

# counting within a specified range
string = "banana republic"
count_a_in_range = string.count("a", 3, 10)
print(count_a_in_range)
2
```