Python Programming

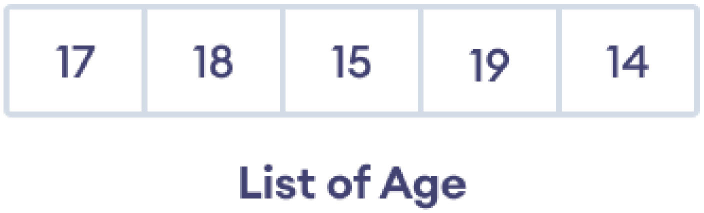
Practical Questions - Database Management

Session 1 - Part 2

LISTS AND TUPLES

In Python, lists are used to store multiple data at once. For example,

Suppose we need to record the ages of **5** students. Instead of creating **5** separate variables, we can simply create a list:



What are Variables?

Python variables are simply **containers for storing data values**. Unlike other languages, such as Java, Python has no command for declaring a variable, so you create one the moment you first assign a value to it. Python variables are simply containers for storing data values.

Create a Python List

A list is created in Python by placing items inside `[]`, separated by commas . For example,

```
# A list with 3 integers
numbers = [1, 2, 5]

print(numbers)

# Output: [1, 2, 5]
```

Here, we have created a list named numbers with **3** integer items.

A list can have any number of items and they may be of different types (integer, float, string, etc.). For example,

```
# empty list
my_list = []

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

Access Python List Elements

In Python, each item in a list is associated with a number. The number is known as a list index.

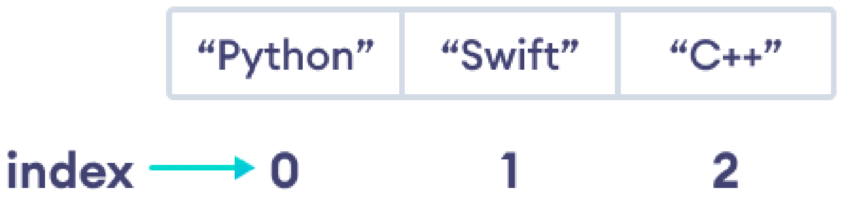
We can access elements of an array using the index number **(0, 1, 2 ...)**

```
languages = ["Python", "Swift", "C++"]

# access item at index 0
print(languages[0])   # Python

# access item at index 2
print(languages[2])   # C++
```

In the above example, we have created a list named languages.



Here, we can see each list item is associated with the index number. And, we have used the index number to access the items.

> **Note:** The list index always starts with **0**. Hence, the first element of a list is present at index **0**, not **1**.

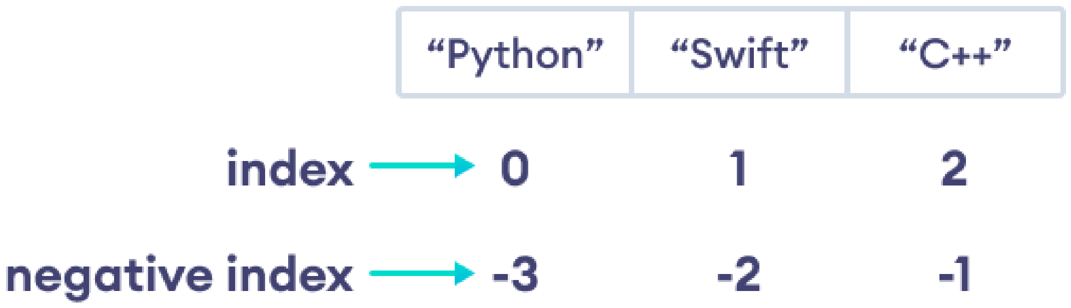
Negative Indexing in Python

Python allows negative indexing for its sequences. The index of **-1** refers to the last item, **-2** to the second last item and so on.

```
languages = ["Python", "Swift", "C++"]

# access item at index 0
print(languages[-1])   # C++

# access item at index 2
print(languages[-3])   # Python
```



Slicing of a Python List

In Python it is possible to access a section of items from the list using the slicing operator `:`, not just a single item. For example,

```
# List slicing in Python

my_list = ['p','r','o','g','r','a','m','i','z']

# items from index 2 to index 4
print(my_list[2:5])

# items from index 5 to end
print(my_list[5:])

# items beginning to end
print(my_list[:])
```

Add Elements to a Python List

Python List provides different methods to add items to a list.

1. Using append()

The append() method adds an item at the end of the list. For example,

```
numbers = [21, 34, 54, 12]

print("Before Append:", numbers)

# using append method
numbers.append(32)

print("After Append:", numbers)
```

2. Using extend()

We use the extend() method to add all items of one list to another. For example,

```
prime_numbers = [2, 3, 5]
print("List1:", prime_numbers)

even_numbers = [4, 6, 8]
print("List2:", even_numbers)

# join two lists
prime_numbers.extend(even_numbers)

print("List after append:", prime_numbers)
```

Change List Items

Python lists are mutable. Meaning lists are changeable. And, we can change items of a list by assigning new values using `=` operator. For example,

```
languages = ['Python', 'Swift', 'C++']

# changing the third item to 'C'
languages[2] = 'C'

print(languages)  # ['Python', 'Swift', 'C']
```

Remove an Item From a List

1. Using del()

In Python we can use the del statement to remove one or more items from a list. For example,

```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# deleting the second item
del languages[1]
print(languages) # ['Python', 'C++', 'C', 'Java', 'Rust', 'R']

# deleting the last item
del languages[-1]
print(languages) # ['Python', 'C++', 'C', 'Java', 'Rust']

# delete first two items
del languages[0 : 2]  # ['C', 'Java', 'Rust']
print(languages)
```

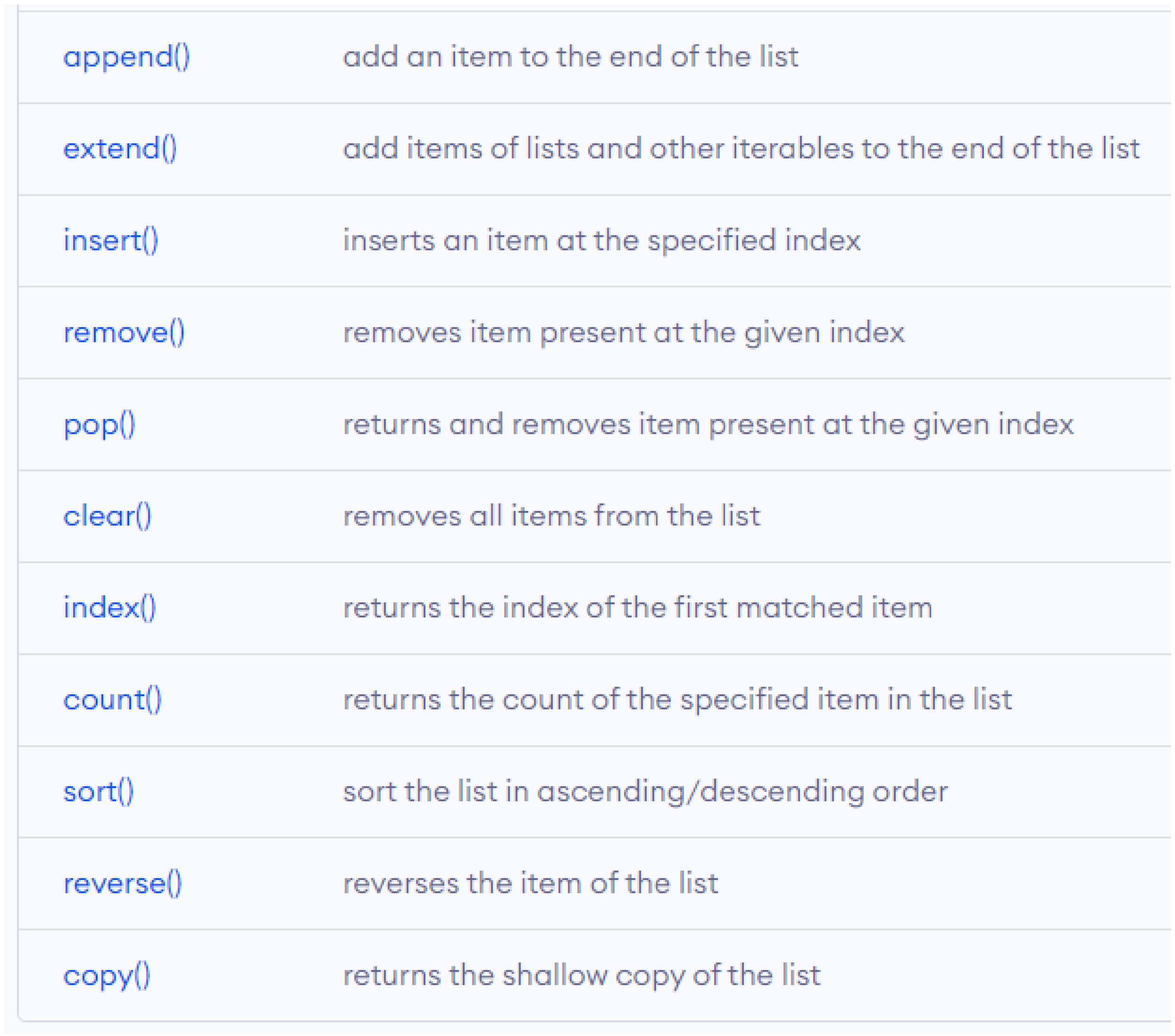
2. Using remove()

We can also use the remove() method to delete a list item. For example,

```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# remove 'Python' from the list
languages.remove('Python')

print(languages) # ['Swift', 'C++', 'C', 'Java', 'Rust', 'R']
```



append()	add an item to the end of the list
extend()	add items of lists and other iterables to the end of the list
insert()	inserts an item at the specified index
remove()	removes item present at the given index
pop()	returns and removes item present at the given index
clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the specified item in the list
sort()	sort the list in ascending/descending order
reverse()	reverses the item of the list
copy()	returns the shallow copy of the list

Iterating through a List

We can use the for loop to iterate over the elements of a list. For example,

```
languages = ['Python', 'Swift', 'C++']

# iterating through the list
for language in languages:
    print(language)
```

Check if an Item Exists in the Python List

We use the `in` keyword to check if an item exists in the list or not.

```python
languages = ['Python', 'Swift', 'C++']

print('C' in languages)    # False
print('Python' in languages)    # True
```

Python List Length

In Python, we use the `len()` function to find the number of elements present in a list. For example,

```python
languages = ['Python', 'Swift', 'C++']

print("List: ", languages)

print("Total Elements: ", len(languages))    # 3
```

Exercise

Question 1

```python
list = [ "New York", "Los Angles", "Boston", "Denver" ]

print(list)     # prints all elements
print(list[0])  # print first element

list2 = [1,3,4,6,4,7,8,2,3]

print(sum(list2))
print(min(list2))
print(max(list2))
print(list2[0])
print(list2[-1])
```

Ans:

Tuples

Reminder: Lists are mutable.

> ## Advantages of Tuple over List in Python
>
> Since tuples are quite similar to lists, both of them are used in similar situations.
>
> However, there are certain advantages of implementing a tuple over a list:
>
> - We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.

> - Since tuples are immutable, iterating through a tuple is faster than with a list. So there is a slight performance boost.
> - Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
> - If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

We can also create tuples without using parentheses:

```
my_tuple = 1, 2, 3
my_tuple = 1, "Hello", 3.4
```

When to use Tuples over Lists:
Well, obviously this depends on your needs. There may be some occasions you specifically do not what data to be changed.
If you have data which is not meant to be changed in the first place, you should choose tuple data type over lists.
But if you know that the data will grow and shrink during the runtime of the application,
you need to go with the list data type.

```
# Creating an empty Tuple
Tuple1 = ()
print("Initial empty Tuple: ")
print(Tuple1)
```

```
# Creating a Tuple
# with the use of string
Tuple1 = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple1)

# Creating a Tuple with
# the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))

# Creating a Tuple
# with the use of built-in function
Tuple1 = tuple('Geeks')
print("\nTuple with the use of function: ")
print(Tuple1)
```

We can use the `type()` function to know which class a variable or a value belongs to.

```
var1 = ("hello")
print(type(var1))  # <class 'str'>

# Creating a tuple having one element
var2 = ("hello",)
print(type(var2))  # <class 'tuple'>

# Parentheses is optional
var3 = "hello",
print(type(var3))  # <class 'tuple'>
```

Here,

- `("hello")` is a string so `type()` returns `str` as class of var1 i.e. `<class 'str'>`
- `("hello",)` and `"hello",` both are tuples so `type()` returns `tuple` as class of var1 i.e. `<class 'tuple'>`

Access Python Tuple Elements

Like a list, each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.

We use the index number to access tuple elements. For example,

1. Indexing

We can use the index operator `[]` to access an item in a tuple, where the index starts from 0.

So, a tuple having **6** elements will have indices from **0** to **5**. Trying to access an index outside of the tuple index range(**6,7,...** in this example) will raise an `IndexError`.

The index must be an integer, so we cannot use float or other types. This will result in `TypeError`.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# accessing tuple elements using indexing
letters = ("p", "r", "o", "g", "r", "a", "m", "i", "z")

print(letters[0])   # prints "p"
print(letters[5])   # prints "a"
```

In the above example,

- `letters[0]` - accesses the first element
- `letters[5]` - accesses the sixth element

---

2. Negative Indexing

Python allows negative indexing for its sequences.

The index of **-1** refers to the last item, **-2** to the second last item and so on. For example,

```
# accessing tuple elements using negative indexing
letters = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(letters[-1])   # prints 'z'
print(letters[-3])   # prints 'm'
```

In the above example,

- `letters[-1]` - accesses last element
- `letters[-3]` - accesses third last element

---

3. Slicing

We can access a range of items in a tuple by using the slicing operator colon `:`

```
# accessing tuple elements using slicing
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# elements 2nd to 4th index
print(my_tuple[1:4])  #  prints ('r', 'o', 'g')
```

```
# elements beginning to 2nd
print(my_tuple[:-7]) # prints ('p', 'r')

# elements 8th to end
print(my_tuple[7:]) # prints ('i', 'z')

# elements beginning to end
print(my_tuple[:]) # Prints ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

```
('r', 'o', 'g')
('p', 'r')
('i', 'z')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Here,

- `my_tuple[1:4]` returns a tuple with elements from index **1** to index **3**.
- `my_tuple[:-7]` returns a tuple with elements from beginning to index **2**.
- `my_tuple[7:]` returns a tuple with elements from index **7** to the end.
- `my_tuple[:]` returns all tuple items.

Note: When we slice lists, the start index is inclusive but the end index is exclusive.

Python Tuple Methods

In Python ,methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # prints 2
print(my_tuple.index('l'))  # prints 3
```

Here,

- `my_tuple.count('p')` - counts total number of `'p'` in my_tuple
- `my_tuple.index('l')` - returns the first occurrence of `'l'` in my_tuple

Iterating through a Tuple in Python

We can use the for loop to iterate over the elements of a tuple. For example,

```
languages = ('Python', 'Swift', 'C++')

# iterating through the tuple
for language in languages:
    print(language)
```

```
Python
Swift
C++
```

Check if an Item Exists in the Python Tuple

We use the `in` keyword to check if an item exists in the tuple or not. For example,

```
languages = ('Python', 'Swift', 'C++')

print('C' in languages)    # False
print('Python' in languages)    # True
```

```
# Python program to show how to create a tuple
# Creating an empty tuple
empty_tuple = ()
print("Empty tuple: ", empty_tuple)

# Creating tuple having integers
int_tuple = (4, 6, 8, 10, 12, 14)
print("Tuple with integers: ", int_tuple)

# Creating a tuple having objects of different data types
mixed_tuple = (4, "Python", 9.3)
print("Tuple with different data types: ", mixed_tuple)

# Creating a nested tuple
nested_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
print("A nested tuple: ", nested_tuple)
```

Exercise

Question 1

```
a = (1,2,3,4,5)
del a
print(a)
```

Ans:

Question 2

```
a = (1,2,1,3,1,3,1,2,1,4,1,5,1,5)
print(a.count(1))
print(a.index(5))
```

Ans:

Question 3

i) Create an empty tuple and print output

ii) Create a tuple with integers: 4, 6, 8, 10, 12, 14 and print outcome

iii) Create a tupe with having objects with different data types: => integer 4, string "Python",and float 9.3 and print output on screen

iv) Create a nested tuple having => "Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6) and print output on screen

- What object is "Python"?   Ans: _ _ _
- What object is (5, 3, 5, 6)? Ans: _ _ _
- What object is {4: 5, 6: 2, 8:2}? Ans: _ _ _