Python Fundamentals (Part3)

Concepts : String, List, Tuple, Dictionary & Set

In this chapter we are going to dive deep into some of the not-so-simple data types of Python.

Strings

What is a String?

A **string** in Python is a sequence of characters enclosed in quotes:

```
str1 = "hello world"
str2 = 'Prime'
```

Strings are **immutable**, meaning once created, their contents can't be changed directly.

`len()` Function

We can use the built-in `len()` function to print the length of a string:

```
word = 'Prime'
print(len(word))        # 5
```

Concatenation

Just like we add numbers, we can concatenate (i.e. join) strings using the `+` operator.

```
str1 = "Apna"
str2 = "College"

word = str1 + " " + str2        # concatenation
print(word)
```

Loop on Strings

```
s = "Python"
for ch in s:     # ch will store individual chars - 'P', 'y', 't' & so on
    print(ch)
```

Indexing in Strings

Index in a sequence (like strings) represents the position value of individual elements i.e. characters in case of a string. So Indexing lets us access individual characters in a string.

Python follows **Zero-based indexing** i.e. first character is at index `0`.

```
s = "Python"
print(s[0])  # 'P'
print(s[3])  # 'h'
print(s[-1]) # 'n' (negative index: last character)
```

Slicing in Strings

Slicing is a powerful feature of Python that lets us access multiple elements at once. We can do slicing in strings & even on other sequences like lists & tuples.
The general syntax for slicing a string is:

```
string[start : end : step]
```
(stop)

Where:

- **start** - index where the slice starts (inclusive). Defaults to `0` if omitted.
- **stop** - index where the slice ends (exclusive). Defaults to the end of the string if omitted.
- **step** - how many indices to move forward each time. Defaults to `1`.

Example:

```
s = "Python"
print(s[0:2])   # 'Py'
print(s[2:])    # 'thon'
print(s[:3])    # 'Py t'
print(s[::2])   # 'Pto' (every second char)
print(s[::-1])  # 'nohtyP' (reversed string)
```

String Formatting

String formatting is the process of creating **dynamic strings** by inserting values from variables or expressions into a predefined string template. This allows for the construction of flexible and informative output based on changing data.
We have multiple ways to format a string, the most modern 2 are:

1. using the `format()` function
2. using `f-strings`

1. Using `.format()`

In this way we use `{}` as placeholders & pass placeholder replacement values in the format function.

```
name = "Rahul"
age = 25

text = "My name is {} and I am {} years old".format(name, age)
print(text)
```

We can also use positional & named placeholders:

```
"Coordinates: {1}, {0}".format("x", "y") # 'Coordinates: y, x'

"Name: {n}, Age: {a}".format(n="Bob", a=30)
```

2. **Using f-strings** (Python 3.6+)
F-strings are concise, readable & will be our preferred way going forward.
In f-strings we prefix the string with `f` and put our variable or expressions inside `{}` .

```
name = "Rahul"
age = 25
text = f"My name is {name} and I am {age} years old"
print(text)
```

You can put any valid Python expression inside `{}` :

```
a = 5
b = 10
print(f"sum of {a} & {b} = {a + b}")
print(f"avg of {a} & {b} = {(a + b) / 2}")
```

Lists

What is a List?

- A **list** is a **collection of items** in Python.
- Items in a list are **ordered, changeable (mutable), and can contain duplicates**.
- Lists can hold any data type: numbers, strings, other lists, etc.
- Lists are written using square brackets `[]`

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
print(type(my_list))     # <class 'list'>

my_list2 = [10, "Hello", 3.14, True, 10]     # heterogenous list
print(my_list2)
```

List Characteristics

1. **Ordered** – Items have a defined order and can be accessed by index.
2. **Mutable** – Items can be changed after the list is created.
3. **Allows duplicates** – Same value can appear more than once.
4. **Heterogeneous** – Can contain different data types.

List Indexing

Index in list is the position value of an item. Index starts from `0`.
We can use index to access elements, modify the list or even slice it.

Access Elements

```
my_list = ["apple", "banana", "cherry"]
print(my_list[0])  # apple
print(my_list[1])  # banana
print(my_list[-1]) # cherry (last element)
```

Modify Elements

```
my_list = [1, 2, 3, 4]
my_list[0] = 10
print(my_list) # [10, 2, 3, 4]
```

Slicing - Slicing in lists is same as slicing in strings.
The general syntax for slicing a list is:

```
list[start:stop:step]
```
(stop → end)

- **start** - inclusive
- **end** - exclusive
- **step** - optional (default = 1)

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

# Simple Slice
print(numbers[2:5])   # Output: [2, 3, 4]

print(numbers[:4])    # Output: [0, 1, 2, 3] (from start to index 3)
print(numbers[5:])    # Output: [5, 6, 7, 8, 9] (from index 5 to end)
print(numbers[:])     # Output: [0,1,2,3,4,5,6,7,8,9] (copy of the whole list)

# using STEP
print(numbers[::2])  # Output: [0, 2, 4, 6, 8] (every2nd element)
print(numbers[1::3]) # Output: [1, 4, 7] (start at 1, every 3rd element)

# NEGATIVE slice
print(numbers[-5:-2]) # Output: [5, 6, 7] (negative indexing from end)
```

List Methods

We have a lot of useful functions that are associated with lists. Let's have a look at some of them:

1. `len()` - returns the number of elements in a list i.e. length of the list.
2. `append(element)` - adds an elements to the end.
3. `insert(element, idx)` - adds at a specific position.
4. `sort()` - sorts in ascending/alphabetical order.
5. `reverse()` - flips the order of elements.

```python
nums = [5, 2, 9]
print(len(nums)) # 3

nums.append(7)
print(nums) # [5, 2, 9, 7]

nums.insert(1, 4)
print(nums) # [5, 4, 2, 9, 7]

nums.sort()
print(nums) # [2, 4, 5, 7, 9]

nums.reverse()
print(nums) # [9, 7, 5, 4, 2]
```

There are many more methods associated with lists. It doesn't make a lot of sense to cover all of them theoretically here, so we'll be covering them whenever we use them in coming chapters.

Loops on Lists

We use the classical `for` loop to traverse the entire list element-wise.

The most common way is using a loop in lists is to print elements.

```
numbers = [10, 20, 30, 40, 50]

for num in numbers:
    print(num)
```

Linear Search

A linear search checks each element one by one to find a target (x).

```
numbers = [5, 12, 7, 3, 18, 9]
x = 18
idx = 0

for num in numbers:
    if num == target:
        print(f"{x} found at index={idx}")
        break
~~idx += 1~~
    idx += 1
```

In the above code, we are assuming that `x` always exists in the list.

Tuples

What is a Tuple?

A **tuple** in Python is an ordered, immutable collection of items.
Example:

```
tup = (10, 20, 30)

print(tup)
print(type(tup)) # < class 'tuple'>

empty_tu ple = ()   # empty_tuple

single_element_tuple = (42,)
```

Tuple Characteristics

1. Ordered
2. **Immutable** – Items cannot be changed after the tuple is created.

3. Allows duplicates
4. Heterogeneous

Tuples are very similar to lists. Some of the differences are that tuples are immutable & because of this immutability they are also faster.

Indexing & Slicing (Same as Lists)

```
t = (10, 20, 30, 40)

print(t[0])   # 10
print(t[-1])  # 40
print(t[1:3]) # (20, 30)
```

Loops on Tuples (Same as Lists)

```
t = (10, 20, 30, 40)

for val in t:
    print(val)
```

Using loops to calculate **sum** of all elements in the tuple:

```
t = (5, 15, 25)

sum = 0
for val in t:
    sum val

print("Sum:", sum)
```

Tuple Methods

Let's have a look at some of tuple methods:

1. `index(val)` - returns index of first occurrence for any val
2. `count(val)` - returns total count of occurrence for any val

```
t = (1, 2, 2, 3, 5)

print(t.index(2)) # 1
print(t.count(2)) # 3
```

Dictionaries

What is a Dictionary?

A dictionary is an unordered, mutable collection of **key-value pairs**.
Example:

```
my_dict = {
    "name": "Shradha",
    "age": 30,
    "city": "Delhi"
}
```

Dictionary Characteristics

1. Dictionary Keys must be **unique**
2. Dictionary Keys must be **immutable** (e.g., strings, numbers, tuples)
3. Dictionary is mutable.
4. Dictionary values can be anything (lists, other dictionaries, etc.)
5. Unordered (although in modern Python, dictionaries preserve insertion order)

Accessing values (using key & `[]`)

```
student = {"name": "Bob", "age": 20}
print(student["name"])  # Bob
```

Dictionary Methods

Let's have a look at some of the important dictionary methods:

1. `keys()` - returns all keys
2. `values()` - returns all values
3. `items()` - returns key–value pairs as tuples
4. `get(key)` - a safer way to access value of a particular key. Instead of throwing an error it returns `None` if key doesn't exist
5. `update(new_item)` - adds a new item to the dictionary

```
d = {
    "name": "Shradha,
    "subjects": ["math", "science", "physics"],
    "cgpa": 9.5
}

print(d.keys()) # dict_keys
print(d.values()) # dict_values
print(d.items()) # dict_items

print(d.get("cgpa2")) # return None as no such key as "cgpa2"

new_item = {"city": "Delhi"}
print(d.update(new_item))
print(d)
```

Loops on Dictionary

We can loop through using key-pair values:

```
d = {
    "name": "Shradha,
    "subjects": ["math", "science", "physics"],
    "cgp a": 9.5
}

for key, value in d.items():
    print(key, value)
```

Sets

What is a Set?

A set is an unordered collection of **unique elements**.
Example:

```
my_set = {1, 2, 2, 2, 3}

print(my_set)           # {1, 2, 3}
print(type(my_set))
print(len(my_set))      # 3


empty_set = set()
```

Set Characteristics

1. Sets can only have **unique** elements.
2. They are **unordered** - no indexing or slicing.
3. They are **mutable** - we can add or remove elements.
4. Set elements must be **immutable** (like strings, numbers, tuples).

Sets are often used when we need uniqueness or mathematical set operations.

Set Methods

Let's have a look at some of the important set methods:

1. `add(val)` - adds an element to set
2. `remove(val)` - removes an element (raises error if not found)
3. `clear()` - removes all elements
4. `pop()` - removes and returns a random element (since sets are unordered)
5. `s1.union(s2)` - returns new union (union is collection of all unique values in both sets)
6. `s1.ntersection(s2)` - returns new union (intersection is collection of all common & unique values in both sets)

```
s = {10, 20, 30}

s.add(40)       # {10, 20, 30, 40}
print(s)

s.remove(10)    # {20, 30, 40}
print(s)

print(s.pop())  # can be any value

s.clear()
print(s)        # set() - empty set

# Union & Intersection
A = {1, 2, 3}
B = {3, 4, 5}

print(A.union(B))         # {1, 2, 3, 4, 5}
print(A.intersection(B))  # {3}
```

> *Keep learning & Keep exploring!*