02 Python Fundamentals

01 Strings

- immutable data structure

capitalize()

- Converts the first character of the string to uppercase

```
text = "hello world"
print(text.capitalize())  # Output: "Hello world"
```

title()

- Converts the first character of each word to uppercase

```
text = "hello world"
print(text.title())  # Output: "Hello World"
```

lower()

```
text = "Hello World"
print(text.lower())  # Output: "hello world"
```

upper()

```
text = "Hello World"
print(text.upper())  # Output: "HELLO WORLD"
```

swapcase()

- Swaps the case of all characters in the string

```
text = "Hello World"
print(text.swapcase())  # Output: "hELLO wORLD"
```

strip()

- Removes leading and trailing whitespace

```
text = "   Hello World   "
print(text.strip())  # Output: "Hello World"
```

- lstrip
- rstrip

replace()

- Adds an element to the end of the list

```
text = "Hello World"
print(text.replace("World", "Universe"))
# Output: "Hello Universe"
```

split()

- Splits the string into a list of substrings based on a delimiter (default is whitespace)

```
text = "Hello World"
print(text.split())  # Output: ["Hello", "World"]
```

join()

- Joins a list of strings into a single string with a specified delimiter

```
words = ["Hello", "World"]
print(" ".join(words))  # Output: "Hello World"
```

find()

- Returns the lowest index of the substring if found, otherwise -1

```
text = "Hello World"
print(text.find("World"))  # Output: 6
print(text.find("Universe"))  # Output: -1
```

index()

- Returns the lowest index of the substring if found, otherwise raises a ValueError

```
text = "Hello World"
print(text.index("World"))  # Output: 6
# print(text.index("Universe"))  # Raises ValueError
```

count()

```
text = "Hello World"
print(text.count("l"))  # Output: 3
```

startswith() & endswith()

- Returns `True` if the string starts/ ends with the specified prefix, otherwise `False`

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

isdigit() & isalpha() & isalnum() & islower() & isupper()

- Returns `True` if the string follows the condition, otherwise `False`
- isspace()

format()

- Formats specified values in a string

```
text = "Hello, {}. Welcome to {}!"
formatted_text = text.format("Alice", "Wonderland")
print(formatted_text)
# Output: "Hello, Alice. Welcome to Wonderland!"
```

zfill()

- Pads a numeric string with zeros on the left, to fill the specified width

```
text = "42"
padded_text = text.zfill(5)
print(padded_text)  # Output: "00042"
```

---

02 List

---

append()

- Adds an element to the end of the list

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

extend()

- Extends the list by appending all the elements from the iterable

```
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)  # Output: [1, 2, 3, 4, 5]
```

insert()

- Inserts an element at a specified position

```
my_list = [1, 2, 3]
my_list.insert(1, 4)
print(my_list)  # Output: [1, 4, 2, 3]
```

remove()

- Removes the first item from the list whose value is equal to the specified value

- Raises a ValueError if the item is not found

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
print(my_list)  # Output: [1, 3, 2]
```

pop()

- Removes and returns the item at the specified position
- If no index is specified, removes and returns the last item

```
my_list = [1, 2, 3]
item = my_list.pop()
print(item)  # Output: 3
print(my_list)  # Output: [1, 2]
```

clear()

- Removes all items from the list

```
my_list = [1, 2, 3]
my_list.clear()
print(my_list)  # Output: []
```

index()

- Returns the index of the first item whose value is equal to the specified value
- Raises a ValueError if the item is not found

```
my_list = [1, 2, 3, 2]
index = my_list.index(2)
print(index)  # Output: 1
```

count()

- Returns the number of times the specified value appears in the list

```
my_list = [1, 2, 3, 2]
count = my_list.count(2)
```

```
print(count)  # Output: 2
```

sort()

- Sorts the list in ascending order by default
- You can specify the `reverse` parameter to sort in descending order

```
my_list = [3, 1, 2]
my_list.sort()
print(my_list)  # Output: [1, 2, 3]

my_list.sort(reverse=True)
print(my_list)  # Output: [3, 2, 1]
```

reverse()

- Reverses the elements of the list in place

```
my_list = [1, 2, 3]
my_list.reverse()
print(my_list)  # Output: [3, 2, 1]
```

copy()

- Returns a shallow copy of the list

```
my_list = [1, 2, 3]
new_list = my_list.copy()
print(new_list)  # Output: [1, 2, 3]
```

list() Constructor

- Creates a new list

```
# Using the list constructor
my_list = list([1, 2, 3])
print(my_list)  # Output: [1, 2, 3]
```

List Comprehension

- List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc

```
squares = [x**2 for x in range(10)]
print(squares)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

---

03 Tuple

- immutable
- Faster than lists
- Can be used as keys in dictionaries, while lists can't
- To create a tuple with a single element, we have to include the final comma, Without the comma, Python treats it as a string in parentheses

```
t = ("tuples", "are", "immutable")
print(t[0]) # Output: 'tuples'
```

count()

- Returns the number of times a specified value appears in the tuple

```
my_tuple = (1, 2, 3, 2, 4, 2)
count = my_tuple.count(2)
print(count)  # Output: 3
```

index()

- Returns the index of the first occurrence of the specified value
- Raises a ValueError if the value is not found

```
my_tuple = (1, 2, 3, 2, 4, 2)
index = my_tuple.index(3)
print(index)  # Output: 2
```

Slicing

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[1:3])  # Output: (2, 3)
print(my_tuple[:4])   # Output: (1, 2, 3, 4)
```

Concatenation

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5)
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)  # Output: (1, 2, 3, 4, 5)
```

Repetition

```
my_tuple = (1, 2, 3)
repeated_tuple = my_tuple * 2
print(repeated_tuple)  # Output: (1, 2, 3, 1, 2, 3)
```

Tuple Unpacking

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

---

04 Dictionary

---

dict() Constructor

- Creates a new dictionary

```
# Using the dict constructor
my_dict = dict(name='Alice', age=25)
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

accessing values

- Access the value associated with a specific key

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict['name'])  # Output: 'Alice'
print(my_dict.get('age'))  # Output: 25
```

keys()

- Returns a view object that displays a list of all the keys

```
my_dict = {'name': 'Alice', 'age': 25}
keys = my_dict.keys()
print(keys)  # Output: dict_keys(['name', 'age'])
```

values()

- Returns a view object that displays a list of all the values

```
my_dict = {'name': 'Alice', 'age': 25}
values = my_dict.values()
print(values)  # Output: dict_values(['Alice', 25])
```

items()

- Returns a view object that displays a list of a dictionary's key-value tuple pairs

```
my_dict = {'name': 'Alice', 'age': 25}
items = my_dict.items()
print(items)
# Output: dict_items([('name', 'Alice'), ('age',25)])
```

update()

- Updates the dictionary with the elements from another dictionary object or from an iterable of key-value pairs

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict.update({'age': 26, 'city': 'Wonderland'})
```

```
print(my_dict)
# Output: {'name': 'Alice', 'age': 26, 'city': 'Wonderland'}
```

pop()

- Removes and returns the value for the specified key
- If the key is not found, it raises a KeyError

```
my_dict = {'name': 'Alice', 'age': 25}
age = my_dict.pop('age')
print(age)  # Output: 25
print(my_dict)  # Output: {'name': 'Alice'}
```

popitem()

- Removes and returns the last inserted key-value pair as a tuple

```
my_dict = {'name': 'Alice', 'age': 25}
item = my_dict.popitem()
print(item)  # Output: ('age', 25)
print(my_dict)  # Output: {'name': 'Alice'}
```

setdefault()

- Returns the value of the specified key
- If the key does not exist, insert the key with the specified value

```
my_dict = {'name': 'Alice'}
age = my_dict.setdefault('age', 25)
print(age)  # Output: 25
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

clear()

- Removes all elements from the dictionary

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

copy()

- Returns a shallow copy of the dictionary

```
my_dict = {'name': 'Alice', 'age': 25}
new_dict = my_dict.copy()
print(new_dict)  # Output: {'name': 'Alice', 'age': 25}
```

fromkeys()

- Creates a new dictionary from the given sequence of keys, with the specified value for all keys

```
keys = ['name', 'age', 'city']
value = None
new_dict = dict.fromkeys(keys, value)
print(new_dict)
# Output: {'name': None, 'age': None, 'city': None}
```

in operator

- Checks if a key exists in the dictionary

```
my_dict = {'name': 'Alice', 'age': 25}
print('name' in my_dict)  # Output: True
print('city' in my_dict)  # Output: False
```

Dictionary Comprehension

```
keys = ['m', 'a', 'l', 'y']
values = [2, 4, 3, 2]
myDict = { k:v for (k,v) in zip(keys, values)}
print(myDict)

# {'m': 2, 'a': 4, 'l': 3, 'y': 2}
```

---

05 Set

---

add()

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

remove()

- Removes the specified element from the set
- Raises a KeyError if the element is not found

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4}
```

discard()

- Removes the specified element from the set if it is present
- Does not raise an error if the element is not found

```
my_set = {1, 2, 3, 4}
my_set.discard(3)
my_set.discard(5)  # No error raised
print(my_set)  # Output: {1, 2, 4}
```

clear()

```
my_set = {1, 2, 3, 4}
my_set.clear()
print(my_set)  # Output: set()
```

copy()

```
my_set = {1, 2, 3, 4}
new_set = my_set.copy()
print(new_set)  # Output: {1, 2, 3, 4}
```

pop()

- Removes and returns an arbitrary element from the set
- Raises a KeyError if the set is empty

```
my_set = {1, 2, 3, 4}
element = my_set.pop()
print(element)  # Output: (An arbitrary element from the set)
print(my_set)  # Output: Set without the popped element
```

union()

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

intersection()

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}
```

difference()

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

symmetric_difference()

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

update()

- union update

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set1.update(set2)
print(set1)  # Output: {1, 2, 3, 4, 5}
```

- intersection_update()
- difference_update()
- symmetric_difference_update()

Functions

Abstraction Mechanism & Top-down design

```
def main():
    name = input("Enter your name: ")
    print(greet(name))

def greet(name):
    return f"Hello, {name}!"

main()
```

Recursive Functions

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))  # 120
```

kwargs

- kwargs allows you to pass keyworded variable length of arguments to a function

```
def test_kwargs(farg, **kwargs):
        print ("formal arg:", farg)
        for key in kwargs:
                print ("another keyword arg: %s: %s" % (key, kwargs[key]))

test_kwargs(farg=1, myarg2="two", myarg3=3)
```

args

- is used to send a non-keyworded variable length argument list to the function

```
def test_args(first, second, third, *args):
        print("First: %s" % first)
        print("Second: %s" % second)
        print("Third: %s" % third)
        print("And all the rest... %s" % list(args))

test_args('one','two','three','1','2','3','4')
```

Lambda functions

- A lambda is an anonymous function
- It has no name of its own, but contains the names of its arguments as well as a single expression
- `lambda <argname-1, ..., argname-n>: <expression>`

```
# Lambda function to add two numbers
add = lambda x, y: x + y

result = add(3, 5)
print(result)  # Output: 8
```

Higher order functions

- Higher-order function expects a function and a set of data values as arguments
- Separates the task of transforming each data value from the logic of accumulating the results

01 map

- his process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results

```
l = ['sat', 'bat', 'cat', 'mat']
test = list(map(list, l))
print(test)

# [['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', 'a', 't']]
```

02 filtering

- A function called a predicate is applied to each value in a list. If the predicate returns True , the value passes the test and is added to a filter object (similar to a map object). Otherwise, the value is dropped from consideration

```
def odd(n):
        return n % 2 == 1

list(filter(odd, range(10)))

# [1, 3, 5, 7, 9]
```

03 Reduce

- Take a list of values and repeatedly apply a function to accumulate a single data value

```
from functools import reduce
def multiply(x, y): return x * y
data = [1, 2, 3, 4]
print(reduce(add, data))
# 24
```

Text files

- Software object that stores data on a permanent medium

```
f = open("myfile.txt", 'w')
f.readlines()
f.write("First line.\nSecond line.\n")

# Writing to a file
```

```
with open('example.txt', 'w') as file:
    file.write('Hello, world!')

# Reading from a file
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

f = open('workfile', 'rb+')
f.write(b'0123456789abcdef')    # 16

f.seek(5)                       # Go to the 6th byte in the file - 5
f.read(1)                       # b'5'
f.seek(-3, 2)                   # Go to the 3rd byte before the end - 13
f.read(1)                       # b'd'
```

Date & Time

```
from datetime import datetime

now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S")) # '2024-07-02 10:29:56'
```

Working with JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format
- Python provides a built-in package called `json` to work with JSON data

```
import json

# Converting Python object to JSON string
data = {
                name': 'Alice',
                'age': 25
}
json_data = json.dumps(data)
print(json_data)

# Converting JSON string to Python object
json_str = '{"name": "Bob", "age": 30}'
data = json.loads(json_str)
print(data)
```