Sets in Python

Sets -

- are Python containers
- are an unordered sequence of mixed data (immutable objects)
- encloses elements in a pair of curly brackets, separated by commas
- mutable
- do not allow duplicates
- allow set operations on data

Creating a set

In [ ]:
```
s = {10, 20, 30, 40, 50}
print(s)
```

Empty Set

In [ ]:
```
set()
```

Note - bool() of empty set is always False

Add elements to Set

set.add(`obj`)

- adds a new element to the set

In [ ]:
```
s.add("abcd")
print(s)
```

set.update(`seq`)

- takes a sequence object as a parameter and adds all the elemnts from the sequence to the set

In [ ]:
```
s.update([1, 2, 3, 4])
print(s)
```

Remove elememnt from sets

pop()

- removes a random element from the set

In [ ]:
```
s.pop()
print(s)
```

remove(`obj`)

- removes a specified elemnt from the set, givevs error if the element is not present in the set

In [ ]:
```
s.remove("abcd")
print(s)
```

In [ ]:
```
s.remove("abcd")
print(s)
```

discard(`obj`)

- removes the specified element from the set, it will not give any error if element is not present.

In [ ]:
```
print(s.discard("abcd"))
```

Built-in functions on Sets

- **len()** - returns length of the sets
- **min(), max()** - returns minimum and maximum element from the set
- **sorted()** - sorts the elements of the set and returns a list
- **sum()** - applicable to only numeric sets, returns summation of all the elements int the set

In [ ]:
```
set_a = {10, 20, 30, 40, 50, 20}
```

In [ ]:
```
len(set_a)
```

In [ ]:
```
min(set_a)
```

In [ ]:
```
sum(set_a)
```

In [ ]:
```
sorted(set_a)
```

Operations on Sets

- Iteration
- Membership
- Set Operations

- Union | Intersection | Difference | Symmetric Difference
- Disjoint sets
- Subsets and Supersets

Union | Intersection | Difference | Symmetric Difference

In [ ]:
```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
```

In [ ]:
```
set1 | set2 # union
set1.union(set2)
```

In [ ]:
```
set1 & set2 # intersection
set1.intersection(set2)
```

In [ ]:
```
set1 ^ set2
set1.symmetric_difference(set2)
```

In [ ]:
```
set1 - set2
set1.difference(set2)
```

Disjoint set

- if the two sets have no common elements

In [ ]:
```
set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}

set1.isdisjoint(set2)
```

Subset | Superset

- If all elemenets of set1 are present in set2 then,
  - set1 is subset of set2
  - set2 will be superset of set1

In [ ]:
```
set1 = {1, 2, 3, 4, 5}
set2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

set1.issubset(set2)
```

In [ ]:
```
set2.issuperset(set1)
```

Dictionary in Python

Dictionaries are -

- are Python containers
- are an unordered sequence of mixed data
- Encloses elements in a pair of curly brackets
- elements are stored in the form of {key : value} pairs separated by commas
- keys are always unique and immutable
- values need not be unique and can be of any type
- mutable

Empty Dictionary

In [ ]:
```
{}

dict()
```

Note - bool() of empty dict is always False

Creating a dictionary

Ex. Create a dictionary consisting of country names and their currencies

In [ ]:
```
countries = {"India" : "INR", "USA" : "USD"}
print(countries)
```

Retriving elements from a Dictionary

Ex. Print currency for "India"

In [ ]:
```
countries["India"]
```

Ex. Print currency for "Japan"

In [ ]:
```
countries["Japan"]
```

dict.get() - returns the value of the item with the specified key

dictionary.get(keyname, value)

- keyname - Required. The keyname of the item you want to return the value from
- value - Optional. A value to return if the specified key does not exist. Default value None

In [ ]:
```
countries.get("India")
```

In [ ]:
```
print(countries.get("Japan"))
```

In [ ]:
```
print(countries.get("Japan", "Country not present"))
```

Adding new element to dictionary

Ex. Add Japan and its currency to dictionary

In [ ]:
```
countries["Japan"] = "Yen"
print(countries)
```

Modifying dictionary

Ex. Modify the currency for USA as "$"

In [ ]:
```
countries["USA"] = "$"
print(countries)
```

Updating a dictionary

dict.update(`new_dict`) - inserts the specified items to the dictionary

Ex. Add contents from new_country dictionary to countries

In [ ]:
```
new_dict = {"Indonesia" : "IDR", "Singapore" : "SGD", "Thailand" : "Bhat"}
countries.update(new_dict)
print(countries)
```

Remove element from dictionary

dict.pop(`key`)

- removes the specified key an its value from the dictionary

In [ ]:
```
countries.pop("USA")
```

dict.popitem()

- randomly removes a key-value pair from dictionary

In [ ]:
```
countries.popitem()
```

dict.clear()

- removes all the pairs from the dictionary

In [ ]:
```
countries.clear()
```

Dictionary Methods

- dict.keys()
- dict.values()
- dict.items()

In [ ]:
```
employees = {'Jane': 70000, 'Rosie': 90000, 'Mary': 40000, 'Sam': 55000, 'George':
employees.keys()
```

In [ ]:
```
employees.values()
```

Ex. Is there any employee having Salary = 55000?

In [ ]:
```
55000 in employees.values()
```

Ex. WAP to create a dictionary where keys are employee codes starting from 101 and its values are the employee names

In [ ]:
```
names = ['Jane', 'Rosie', 'Mary', 'Sam', 'George']
dict(enumerate(names, start = 101))
```

Ex. WAP to create a dictionary combining the following two lists where name is key and marks as value

In [ ]:
```
names = ['Jane', 'Rosie', 'Mary', 'Sam', 'George']
salary = [70000, 90000, 40000, 55000, 76000]
dict(zip(names, salary))
```

In [ ]:
```
dict(enumerate(zip(names, salary), start = 101))
```

Comprehensions in Python

`Comprehensions` are an elegant way to define and create mutable data structures like lists, sets, dictionary based on existing sequences Syntax –

```
[<expression> for <var> in <sequence> if <condition>]
```

1. Identify the sequence
2. Identify condition if any
3. Expression
4. Mutable datastructure

Ex. WAP to generate a list of squares of number in range of 1-10

In [ ]:
```
[i**2 for i in range(1, 11)]
```

Ex. WAP to create a list of sqaures of even number in range of 1-10

In [ ]:
```
[i**2 for i in range(1, 11) if i % 2 == 0]
```

Ex. WAP to create a dict of number from 1-10 as keys and their squares as values

In [ ]:
```
{i : i**2 for i in range(1, 11)}
```

Ex. WAP to create a dict of number from 1-10 as keys and their type (even or odd) as values

In [ ]:
```
{i : "even" if i%2 == 0 else "odd" for i in range(1, 11)}
```

Ex. WAP to add 7% service tax to all the values in the "sales" list

In [ ]:
```
sales = [290, 500, 800, 650]
[i*1.07 for i in sales]
```

Ex. WAP to sum all the values in the "sales" tuple

In [ ]:
```
sales = ("$290", "$500", "$800", "$650")
sum([int(i.replace("$", "")) for i in sales])
```

In [ ]:
```
sales = ("$290", "$500", "$800", "$650")
sum(int(i.replace("$", "")) for i in sales)
```

Ex. WAP to create a dict of names and the total marks(percentage) of each student.

In [ ]:
```
names = ['Jane', 'Rosie', 'Mary', 'Sam', 'George']
marks = ([70, 65, 32], [90, 76, 98], [40, 55, 78], [50, 87, 67], [76, 72, 89])

percentage = [f"{round(sum(m)/len(m), 2)}%" for m in marks]
percentage
```

In [ ]:
```
dict(zip(names, percentage))
```

In [ ]:
```
a, b, c = 2, 3, 5

f"addition of {a} and {b} is {c}"
```

Functions in Python

A function is set of statements that take input in the form of parameters, performs computation on the input and returns a result in the form of return statement

Syntax –

def `function-name` (`parameters if any`):

 # function code

`return` statement

Note : It is a best practice to avoid usage of input() and print() functions in a function definition

WAF to calculate factorial of a number

In [ ]:
```
def factorial(num) :
    if type(num) == int :
        fact = 1
        for i in range(num, 1, -1):
            fact *= i

        return fact
```

In [ ]:
```
factorial("abc")
```

Function Arguments

- Required Positional Arguments
- Default Arguments
- Variable length Arguments
- Key-word Arguments
- Variable-length Keyword Arguments

Required Positional Argument

In [ ]:
```
def demo(name, age) :
    print("Name - ", name)
    print("Age - ", age)

demo("Jane", 30)
```

In [ ]:
```
demo("Jane")
```

In [ ]:
```
demo(30, "Jane")
```

In [ ]:
```
help(list.insert)
```

In [ ]:
```
lst = [10, 20, 30, 40, 50]
lst.insert(2, 3)
lst
```

In [ ]:
```
lst = [10, 20, 30, 40, 50]
lst.insert(3, 2)
lst
```

In [ ]:
```
lst = [10, 20, 30, 40, 50]
lst.insert("abcd", 2)
lst
```

In [ ]:
```
help(str.casefold)
```

In [ ]:
```
"A" in "aeiou"
```

In [ ]:
```
"A".casefold() in "aeiou"
```

Default Argument

In [ ]:
```
help(str.replace)
```

In [ ]:
```
strg = "Mississippi"
strg.replace("i", "*")
```

In [ ]:
```
strg = "Mississippi"
strg.replace("i", "*", 2)
```

In [ ]:
```
help(enumerate)
```

Variable-length Argument

In [ ]:
```
def demo(name, *args, age = 18):
    print("Name - ", name)
    print("Age - ", age)
    print("args - ", args)

demo("Jane", 50, 60, 70, 80, 19)
```

Key-word Argument

In [ ]:
```
demo("Jane", 50, 60, 70, 80, age = 19)
```

Variable-length Keyword Argument

In [ ]:
```
def demo(name, *args, age = 18, **kwargs):
    print("Name - ", name)
    print("Age - ", age)
    print("args - ", args)
    print("kwargs - ", kwargs)

demo("Jane", 50, 60, 70, 80, age = 19, gender = "F", mob = 98765432)
```

- `*` - All the arguments after `*` must be passed as keyword-only
- `/` - All the arguments before `/` must be passed as positional-only

In [ ]:
```
def demo(name, age):
    print(name, age)
```

```
demo("Jane", 30)  # Positional-only
demo(name = "Jane", age = 30)  # keyword-only
```

In [ ]:
```
def demo(name, age, /):
    print(name, age)

demo("Jane", 30)
demo(name = "Jane", age = 30)  # Error
```

In [ ]:
```
def demo(name, /, age):
    print(name, age)

demo("Jane", 30)
demo("Jane", age = 30)
demo(name = "Jane", age = 30)  # Error
```

In [ ]:
```
def demo(name, *, age):
    print(name, age)

demo("Jane", age = 30)
demo(name = "Jane", age = 30)
demo("Jane", 30)   # Error
```

Lambda Function

- A lambda function is also called as an anonymous function as it is a function that is defined without a name.
- A lambda function behaves similar to a standard function except it is defined in one-line.
- It is defined using a lambda key-word.
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.
- Lambda functions can be used wherever function objects are required.
- Syntax of Lambda Function –

  lambda *parameters* : *expression*

Write a lambda function to return addition of 2 numbers

In [ ]:
```
lambda a, b : a + b
```

In [ ]:
```
add = lambda a, b : a + b
add(2, 3)
```

Write a lambda function to return square of the number

In [ ]:
```
square = lambda num : num ** 2

square(5)
```

Function Object

- Everything in Python is an object, including functions.
- You can assign them to variables, store them in data structures, and pass or return them to and from other functions
- Functions in Python can be passed as arguments to other functions, assigned to variables or even stored as elements in various data structures.

function definition/implemenation

In [ ]:
```
def func(a, b):  # -> function definition
    if a < b :
        return a
    else:
        return b
```

function call

In [ ]:
```
# function call
var = func(2, 3)
var
```

function object

In [ ]:
```
# function object
var = func
var
```

In [ ]:
```
var(2, 3)
```

In [ ]:
```
x = len
```

In [ ]:
```
x("abcd")
```

Applilcations of Function Object

In [ ]:
```
lst = ["train", "car", "bike", "flight"]
```

Ex. Sort the list alphabetically

In [ ]:
```
sorted(lst)
```

Ex. Sort the list by last character

In [ ]:
```
sorted(lst, key = lambda strg : strg[-1])
```

Ex. Sort the list by number of characters in the word

In [ ]:
```
sorted(lst, key = len)
```

In [ ]:
```
max(lst, key = len)
```

In [ ]:
```
max(lst, key = min)
```

Ex. WAP to display the student details in sorted order of their marks.

In [ ]:
```
students = {"Jane" : 40, "Max" : 50, "Sam" : 45, "Mary" : 70}
sorted(students, key = lambda x : students[x])
```

In [ ]:
```
sorted(students.items(), key = lambda x : x[1])
```

In [ ]:
```
dict(sorted(students.items(), key = lambda x : x[1]))
```

Note -

- There are many functions like sorted which take function object as an argument.
- Use a built-in function if available, else defined a custom function.
- If the logic for custom function is one-liner use lambda function else use standard user-defined function

Regular Expressions

Regular expressions are used for matching text patterns for searching, replacing and parsing text with complex patterns of characters.

Regexes are used for four main purposes -

- To validate if a text meets some criteria; Ex. a zip code with 6 numeric digits
- Search substrings. Ex. finding texts that ends with abc and does not contain any digits
- Search & replace everywhere the match is found within a string; Ex. search "fixed deposit" and replace with "term deposit"
- Split a string at each place the regex matches; Ex. split everywhere a @ is encountered

Raw python string

It is recommended that you use raw strings instead of regular Python strings. Raw strings begin with a prefix, r, placed before the quotes

In [ ]:
```
print("ABC \n PQR")
```

In [ ]:
```
print(r"ABC \n PQR")
```

Importing re module

In [ ]:
```
import re
```

Functions in re Module

The "re" module offers functionalities that allow us to match/search/replace a string

- `re.match()` - The match only if it occurs at the beginning of the string
- `re.search()` - First occurrence of the match if there is a match anywhere in the string
- `re.findall()` - Returns a list containing all matches in the string
- `re.split()` - Returns a list where the string has been split at each match
- `re.sub()` - Replaces one or many matches with a string
- `re.finditer()` - Returns a collectable iterator yielding all non-overlapping matches

In [ ]:
```
text = "Jack and Jill went up the hill"

re.match(r"Jack", text)
```

In [ ]:
```
text = "Jack and Jill went up the hill"

re.search(r"Jill", text)
```

In [ ]:
```
text = "She sells sea shells on the sea shore"

re.findall(r"s", text)
```

In [ ]:
```
text = "She sells sea shells on the sea shore"

re.split(r" ", text)
```

In [ ]:
```
text = "She sells sea shells on the sea shore"

re.sub(r"[aeiou]", "*", text)
```

Basic Characters

- `^` - Matches the expression to its right at the start of a string. It matches every such instance before each line break in the string
- `$` - Matches the expression to its left at the end of a string. It matches every such instance before each line break in the string
- `p|q` - Matches expression p or q

Character Classes

- `\w` - Matches alphanumeric characters: a-z, A-Z, 0-9 and _
- `\W` - Matches non-alphanumeric characters. Ignores a-z, A-Z, 0-9 and _
- `\d` - Matches digits: 0-9
- `\D` - Matches any non-digits
- `\s` - Matches whitespace characters, which include the \t, \n, \r, and space characters
- `\S` - Matches non-whitespace characters
- `\A` - Matches the expression to its right at the absolute start of a string (in single or multi-line mode)
- `\t` - Matches tab character
- `\Z` - Matches the expression to its left at the absolute end of a string (in single or multi-line mode)
- `\n` - Matches a newline character
- `\b` - Matches the word boundary at the start and end of a word
- `\B` - Matches where \b does not, that is, non-word boundary

Groups and Sets

- `[abc]` - Matches either a, b, or c. It does not match abc
- `[a\-z]` - Matches a, -, or z. It matches - because \ escapes it
- `[^abc]` - Adding ^ excludes any character in the set. Here, it matches characters that are NOT a, b or c
- `()` Matches the expression inside the parentheses and groups it
- `[a-zl` - Matches any alphabet from a to z
- `[a-z0-9]` - Matches characters from a to z and O to 9
- `[(+*)]` - Special characters become literal inside a set, so this matches (+ * and)
- `(?P=name)` - Matches the expression matched by an earlier group named "name"

Quantifiers

- `.` - Matches any character except newline
- `?` - Matches the expression to its left O or 1 times
- `{n}` - Matches the expression to its left n times
- `(,m)` - Matches the expression to its left up to m times
- `*` - Matches the expression to its left O or more times
- `+` - Matches the expression to its left 1 or more times
- `{n,m}` - Matches the expression to its left n to m times
- `{n, }` - Matches the expression to its left n or more times

Examples -

Ex. Extract all digits from the text

In [ ]:
```
text = "The stock price was 456 yesterday. Today, it rose to 564"
re.findall(r"\d", text)
```

Ex. Extract all numbers from the text

In [ ]:
```
text = "The stock price was 456 yesterday. Today, it rose to 564"
re.findall(r"\d+", text)
```

Ex. Retrive the dividend from the text

In [ ]:
```
text = "On 25th March, the company declared 17% dividend."
re.findall(r"\d+%", text)
```

Ex. Retrieve all uppercase characters

In [ ]:
```
text = "Stocks like AAPL GOOGL BMW are the preferred ones"
re.findall(r"[A-Z]", text)
```

Ex. Retrive all stock names

In [ ]:
```
text = "Stocks like AAPL GOOGL BMW are the preferred ones"
re.findall(r"[A-Z]+\b", text)
```

Ex. Retrieve the phone numbers with country code only

In [ ]:
```
text = "My number is 65-11223344 and 65-91919191. My other number is 44332211"
re.findall(r"\d+-\d+", text)
```

Ex. Retrieve the phone numbers with or without country code

In [ ]:
```
text = "My number is 65-11223344 and 65-91919191. My other number is 44332211"
re.findall(r"\d+-\d+|\d+", text)
```

Ex. Retrieve the phone numbers without country code

In [ ]:
```
text = "My number is 65-11223344 and 65-91919191. My other number is 44332211"
re.findall(r"\d{3,}", text)
```

Ex. Replace values as given in the dict

In [ ]:
```
text = "Stocks like AAPL GOOGL BMW are the preferred ones"
repl = {"AAPL" : "APPLE", "GOOGL" : "GOOGLE"}
re.sub(r"[A-Z]+\b", "*", text)
```

In [ ]:
```
help(re.sub)
```

In [ ]:
```
obj = re.search(r"[A-Z]+\b", text)
obj.group()
```

In [ ]:
```
repl[obj.group()]
```

In [ ]:
```
re.sub(r"[A-Z]+\b", lambda mtch_obj : repl.get(mtch_obj.group(), mtch_obj.group()),
```

Reading Data from txt file and Object Oriented Programming

- The key function for working with files in Python is the `open()` function.
- The `open()` function takes two parameters; filename, and mode.
- There are four different methods (modes) for opening a file:
  - "r" - Read - Default value. Opens a file for reading, error if the file does not exist
  - "a" - Append - Opens a file for appending, creates the file if it does not exist
  - "w" - Write - Opens a file for writing, creates the file if it does not exist

In [ ]:
```
import os
```

In [ ]:
```
os.getcwd()
```

In [ ]:
```
os.chdir(r"new_path_to_file")
```

Ex. Read file `customers.txt`

In [ ]:
```
file = open(r"customers.txt")
file
```

In [ ]:
```
data = file.readlines()
file.close()
```

Ex. Print numbers of lines in the file

In [ ]:
```
len(data)
```

Learning Objectives -

- Create a class and its constructor (**init**)
- Create object of the class
- adding methods to the class
- adding built-in methods to generate str representation and comparision

- Use comprehension to apply functionality to all elements in the list
- Use map() and filter as application of function object to work on same example
- Application of unpacking of tuples

Ex. Clean data read from the file and store them as Customer objects

In [ ]:
```
# Customer class to store the values
class Customer :
    def __init__(self, c_id, fname, lname, age, prof):
        self.c_id = c_id
        self.name = fname + " " + lname
        self.age = int(age)
        self.profession = prof

    def get_details(self):
        return f"{self.c_id},{self.name},{self.age},{self.profession}"

    def __repr__(self):
        return self.name

    def __lt__(self, obj):
        return self.name < obj.name
```

In [ ]:
```
def clean_data(strg) :
    lst = strg.strip().split(",")
    cust = Customer(*lst)  # here we are unpacking the list into variables
    return cust
```

1. Using Comprehension

In [ ]:
```
customers = [clean_data(i) for i in data]
customers[0].name
```

2. Using map() - application of function object

In [ ]:
```
customers = list(map(clean_data, data))
customers[0].name
```

Ex. Extract information about all `Pilots`.

1. Using Comprehension

In [ ]:
```
pilots = [i for i in customers if i.profession == "Pilot"]
len(pilots)
```

2. Using filter() - application of function object

In [ ]:
```
pilots = list(filter(lambda cust : cust.profession == "Pilot", customers))
len(pilots)
```

Ex. Write names of the pilots to `pilots.txt` file

In [ ]:
```
with open("pilots.txt", "w") as file :
    for p in pilots :
        file.write(p.get_details()+"\n")
print("All data written sucessfully!")
```

In [ ]:
```
    def get_details(self):
        return f"{self.c_id},{self.name},{self.age},{self.profession}"
```

Ex. Display names of customers while displaying the `customers list`

In [ ]:
```
customers[0]
```

In [ ]:
```
customers
```

Ex. Sort the customers list by age

In [ ]:
```
customers[0] < customers[1]
```

In [ ]:
```
sorted(customers, key = lambda cust : cust.age)
```

Example on Inheritance

In [ ]:
```
from abc import ABC, abstractmethod
class Shape(ABC):
    prices = {"red" : 10, "blue" : 20, "green" : 30, "white" : 1}

    def __init__(self, **kwargs):
        self.cal_area()
        self.color = kwargs.get("color", "white") # extract color from kwargs

    @abstractmethod
    def cal_area(self) :
        pass

    def color_cost(self, color = None):
        # extract price for the color
        color = color if color else self.color
        return self.area * Shape.prices.get(color, 1)

class Circle(Shape):
    pi = 3.14

    def __init__(self, radius, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)   # calls constructor of parent class

    def cal_area(self):
        self.area = Circle.pi * (self.radius ** 2)

class Rectangle(Shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
```

```
        self.cal_area()

    def cal_area(self):
        self.area = self.length * self.breadth
```

In [ ]:

In [ ]:

In [ ]: