1. **Python Iterators and Iterables:**

An **Iterator** is an Object that contains a Countable Number of Values and iterator can be iterated upon all the Values.

In Python, an iterator is an object which implements the iterator Protocol, which consist of the methods **\_\_iter()\_\_** and **\_\_next()\_\_** .

**Strings, Lists, Tuples, Dictionaries and sets** are all **Iterable** **Objects**. They are **Iterable Containers** which we can get an iterator from.

Example:

mystr = "HEAD"

myit = iter(mystr)

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

**OUTPUT:**

H

E

A

D

**Looping through an Iterator**:

The **for** loop actually creates an iterator object and executes the next method for each loop.

Example:

mytuple = ("rat","dog","kitten")

for x in mytuple:

print(x)

OUTPUT:

rat

dog

kitten

**How for loop actually works?**

**for element in iterable:**

is actually implemented as:

# create an iterator object from that iterable

**iter\_obj = iter(iterable)**

# infinite loop

**while true:**

**try**:

# get next item

**element = next(iter\_obj)**

**except stopIteration**:

# if stopIteration is raised, break from loop

**Break**

So internally, **for** loop creates an iterator object, **iter\_obj** by calling **iter()** on the iterable.

This **for** loop is actually an **infinite while loop**.

Inside the loop, it calls **next()** to get the next element and executes the body of **for** loop with this value. After all the items exhaust, **stopIteration** is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

**Create an Iterator**:

To Create an object/class as an iterator we have to implement the methods **\_\_iter()\_\_** and **\_\_next()\_\_** to the objects.

The **\_\_iter()\_\_** method allows to do operations but must always **return the iterator object** itself.

The **\_\_next()\_\_** method allows to do operations but must always **return the next item** in the sequence.

Example 1:

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

class MyNumbers:  
  def \_\_iter\_\_(self):  
    self.a = 1  
    return self  
  
  def \_\_next\_\_(self):  
    x = self.a  
    self.a += 1  
    return x  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))

OUTPUT:

1

2

3

4

5

Example 2:

Create an Iterator that returns Top Ten numbers starting form 1

class TopTen:

def \_\_init\_\_(self):  
    self.num = 1

  def \_\_iter\_\_(self):  
       return self  
  
  def \_\_next\_\_(self):

if  self.num <= 10:

value = self.num

self.num += 1  
        return value  
 else:

raise stopIteration

values = TopTen()

print(next(values)) # **Prints 1 and doesn’t print 1 again in next for loop i.e., for loop will print values from 2**

for i in values:  
 print(i)

OUTPUT:

1

2

3

4

5

6

7

8

9

10

2. **Python Generators:**

**Generators are used to fetch one value at a time from more number of values defined in a function using yield keyword.**

1. **Generator Function**:

A Generator Function is defined like a Normal Function, but whenever it needs to generate a value, it does so with **yield** Keyword rather than **return**.

If the body of a **def function contains yield**, the function automatically becomes a **generator function**.

Generator function generates Iterator.

**Example**:

# A generator function that yields 1 for first time, 2 second time and 3 third time

**def** simpleGeneratorFun():

yield 1

yield 2

yield 3

# Driver code to check above generator function

for value in simpleGeneratorFun():

print(value)

OUTPUT:

1

2

3

2. **Generator Object**:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in **for** loop.

# A Python program to demonstrate use of generator object with next()

# A generator function

**def** simpleGeneratorFun():

yield 1

yield 2

yield 3

# x is a generator object

x = simpleGeneratorFun()

# Iterating over the generator object using next

print(x.\_\_next\_\_()); # In Python 2, next()

print(x.\_\_next\_\_());

print(x.\_\_next\_\_());

OUTPUT:

1

2

3

3. **Python Recursive Function**:

If a function calls itself, it is called as **Recursive Function**.

Example:

A Program using Recursive Function to find the Factorial of a Number

def factorial(num):

if num == 1:

return 1

else:

return num \* **factorial(num -1)**  # Here function is calling itself which is called as **Recursive function**

num = 4

print( factorial(num))

OUTPUT:

24

Advantages of Recursive Function:

1. Recursive Functions make the code look clean and simple.

2. A complex task can be broken down into simpler sub-problems using Recursion.

Dis-Advantages of Recursive Function:

1. Some times logic behind recursion is hard to follow through.

2. Recursive calls are more expensive as they take up lot of memory and time.

3. Recursive Functions are hard to debug.

4. **Python RegEx:**

* A Regular Expression (RegEx) is a sequence of characters that defines a search pattern i.e., RegEx is a tool for matching patterns in text.

Example:

**^a…s$**

The above code defines a RegEx Pattern. The Pattern is : **any five letter word starting with** a **and ending with** s.

* Using RegEx, We can **match**, **find** or **replace** text or word in place in strings.
* Python has a **module named re** to work with RegEx.

Example:

import re

pattern = '^a...s$'

test\_str = input("Enter string to search: ")

if **re.match(pattern,test\_str)**:

print("Search Successful")

else:

print("Search Unsuccessful")

OUTPUT 1:

Enter string to search: alias

Search Successful

OUTPUT 2:

Enter string to search: aliaa

Search Unsuccessful

* **Metacharacters** are used to specify Regular Expressions. Metacharacters are characters that are interpreted in a special way by a RegEx Engine. Here’s a list of metacharacters

**[]** **.** **^** **$** **\*** **+** **?** **{}** **()** **\** **|**

**[] – Square Brackets**

* **Square Brackets specifies a set of characters that we wish to match.**
* **[abc] will match if the string we are trying to match will contain any of the a, b or c.**
* **We can also specify a range of characters using – inside the square brackets.**

1. **[a-e] is same as [abcde]**
2. **[1-4] is same as [1234]**
3. **[0-39] is same as [01239]**

* **We can complement (invert) the character set by using caret ^ symbol at the start of the square bracket.**

1. **[^abc] means any character except a or b or c.**
2. **[^0-9] means any Non-Digit Character.**

**. – Period**

**A Period matches any single Character (it can be any character except newline ‘\n’).**

**^ - Caret**

**The Caret symbol ^ is used to check if a string starts with a certain character.**

**$ - Dollar**

**The Dollar symbol $ is used to check if a string ends with a certain character.**

**\* - Star**

**The Star symbol \* matches Zero or More occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X\*' = 0 or X or XX or XXX or XXXX etc**

**+ - Plus**

**The Plus symbol + matches One or More occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X+' = X or XX or XXX or XXXX etc**

**? – Question Mark**

**The Question Mark symbol ? matches Zero or One occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X?' = 0 or X**

**{} – Braces**

**Consider this code : {m,n}. This means at least m, and at most n repetitions of the pattern left to it.**

**Example:**

**'X{4}' = Matches exactly 4X i.e., XXXX**

**a{2,3} --- Matches at least 2 a’s but not more than 3 a’s i.e., aa, aaa**

**abc daat --- 1 match ( at daat)**

**aabc daaaat --- 2 matches ( at aabc daaaat)**

**[0-9]{2,4} --- Matches at least 2 digits but not more than 4 digits**

**12 and 345673 --- 2 matches ( 12 and 345673)**

**1 and 2 ---- No match**

**| - Alternation**

**Vertical bar | is used for Alternation ( or Operator)**

**a | b match any string that contains either a or b.**

**() – Group**

**Parenthesis () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz.**

**\ - Backslash**

**Backslash \ is used to escape various characters including all metacharacters.**

**Example:**

**\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.**

**\s – Matches the Whitespace**

**\S – Matches the Non- Whitespace**

**\d – Matches the single digit**

**\D – Matches the single Non-Digit**

**Functions and Constants to work with RegEx:**

**Python has a module named re to work with RegEx. To use it, we need to import the module**

**import re**

1. **re.findall(): This method returns a list of strings that containing all matches. If the pattern is not found, re.findall() will return empty list.**

**Example: Program to extract numbers from a string**

**import re**

**string = 'Hello 67 Hi 34 Good 45'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**result = re.findall(pattern,string)**

**print(result)**

**OUTPUT:**

**['67', '34', '45']**

1. **re.split(): This method splits the string where there is a match and returns a list of strings where the splits have occurred.**

**Example:**

**import re**

**string = 'Twelve : 12 Eighty Nine : 89.'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**result = re.split(pattern,string) # returns a list of strings where the splits have occurred**

**print(result)**

**OUTPUT:**

**['Twelve : ', 'Eighty Nine : ', '. ']**

**If the pattern is not found, re.split() returns a list containing an empty string.**

**We can pass maxsplit argument to the re.split() method. Its’s the maximum number of splits that will occur.**

**The default value of maxsplit is 0; meaning all possible splits.**

**Example:**

**import re**

**string = 'Twelve : 12 Eighty Nine : 89.'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**# maxsplit = 1 i.e., split only at first occurrence**

**result = re.split(pattern,string,1) # returns a list of strings where the splits have occurred**

**print(result)**

**OUTPUT:**

**['Twelve : ', 'Eighty Nine : 89. ']**

1. **re.sub():**

**Syntax: re.sub(pattern, replace, string)**

**This method returns a string where matched occurrences are replaced with content of the replace variable. If the pattern is not found, re.sub() will return original string.**

**Example:**

**import re**

**string = 'abc 12\**

**def 23 \n f45 6as'**

**pattern = '\s+' # matches one or more occurrences of the whitespace in the string**

**replace = ' ' # Empty String**

**new\_string = re.sub(pattern,replace,string)**

**print(new\_string)**

**OUTPUT:**

**abc12def23f456as**

**We can pass count as fourth parameter to the re.sub() method.**

**re.subn(): This method returns a tuple of 2 items containing the new string and number of substitutions made.**

**Example:**

**import re**

**string = 'abc 12\**

**def 23 \n f45 6as'**

**pattern = '\s+' # matches one or more occurrences of the whitespace in the string**

**replace = ' ' # Empty String**

**new\_string = re.subn(pattern,replace,string)**

**print(new\_string)**

**OUTPUT:**

**('abc12def23f456as', 4)**

1. **re.search(): This method takes 2 arguments: a Pattern and a String. This method looks for the first location where the RegEx pattern produces a match with the string.**

**This search function will search entire string and returns the first occurrence of the pattern.**

**If the search is successful, re.search() returns a match object; if not, it returns None.**

**Syntax: match = re.search(pattern,string,[flag=0])**

**Example:**

**import re**

**string = "Python is fun"**

**# \APython check if 'Python' is at the beginning**

**match = re.search('\APython', string)**

**if match:**

**print("Pattern found inside the string")**

**else:**

**print("Pattern not found")**

**OUTPUT:**

**Pattern found inside the string**

1. **re.match(): This function is used to match any text or word in the strings.**

**This match function will search only at the Beginning of the string.**

**Syntax: match = re.match(pattern,string,[flag=0])**

**Example:**

**import re**

**string = "Pet:Cat I Hate Cats"**

**match = re.match("pet:\w\w\w",string) # \w matches any Alpha-Numeric Character equivalent to [a-zA-Z0-9\_]. Underscore \_ is also considered as alphanumeric character.**

**print(match)**

**print(match.group(0)) # The group() method returns the part of the string where there is a match.**

**OUTPUT:**

**<re.Match object; span=(0, 7), match='pet:cat'>**

**'Pet:Cat'**

**match.start()** returns the **index of the start** of the matched sub-string.

**match.end()** returns the **end index** of the matched sub-string.

**match.span()** returns a tuple containing **start and end index** of the matched part.

**match.re** attribute of a matched object returns a **regular expression object**.

**match.string** attribute returns the **passed String**.

**Using r prefix before RegEx**:

When **r** or **R** **prefix** is used before a regular expression, it means **Raw String**.

Example:

**'\n' is a new line whereas r'\n' means 2 characters: a backslash \ followed by n**

**Example:**

**import re**

**string = "\n and \r are escape sequences"**

**result = re.findall(r'[\n\r]',string)**

**print(result)**

**OUTPUT:**

**['\n', '\r']**

**5. Instance vs. Static vs. Class Methods in Python: The Important Differences**

There are 3 types of Methods in Python:

1. Instance Methods
2. Static Methods
3. Class Methods

**Understanding Decorator Patterns (Decorators):**

Decorator Patterns, or simply called as Decorators, are Functions that we can write them ourselves, or use those included in libraries, or the Python standard library.

Like any function, Decorators perform a task. The difference here is that **Decorators apply Logic or change the behaviour of other functions**.

Decorators are an excellent way to reuse code and can help to separate logic into individual concerns.

The Decorators Pattern is Python’s preferred way of defining static or class methods.

Decorators have to immediately precede a function or class declaration.

Decorators start with **@** sign, and unlike normal methods, we don’t have to put parenthesis on the end.

Example:

**class** DecoratorExample:  
 """ Example Class """  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 @staticmethod  
 **def** example\_function():  
 """ This method is decorated! """  
 print(**'I\'m a decorated function!'**)

de = DecoratorExample()  
de.example\_function()

**1. Instance Methods:**

Instance Methods are most common type of methods in Python Classes because they can access Unique data of their instance.

If we have two objects each created from a Car class, then each object may have different properties like different colours, engine sizes, seats, and so on.

Instance Methods must have **self** as a parameter, but no need to pass this in every time. **self** is another Python special term.

We can use **self** to access any data or methods that may reside in our class. We won’t able to be access them without going through **self**.

Finally, instance methods are the most common, there’s is **NO Decorator needed**. Any method that we create will automatically created as an instance Method.

Example 1:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
 self.name = **'Decorator\_Example'  
  
 def** example\_function(self):  
 """ This method is an instance method! """  
 print(**'I\'m an instance method!'**)  
 print(**'My name is '** + self.name)  
  
de = DecoratorExample()  
de.example\_function()

OUTPUT:

Hello, World!

I'm an instance method!

My name is Decorator\_Example

**Note**: The **name** variable is accessed through **self**. Notice that when **example\_function** is called, we don’t have to pass **self** as an argument– Python does for us.

Example 2:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 **def** example\_function(self,name):  
 """ This method is an instance method! """  
 print(**'I\'m an instance method! with 2 parameters'**)  
 self.name = name  
 print(**'My name is '** + self.name)  
  
de = DecoratorExample()  
de.example\_function(**"Decorator"**)

OUTPUT:

Hello, World!

I'm an instance method! with 2 parameters

My name is Decorator

**Note**: When **example\_function** is called, we passed only **name = "Decorator"** as an argument and we don’t have to pass **self** – Python does for us.

**2. Static Methods:**

Static Methods are methods that are related to a class in some way, but no need to access any class-specific data and even no need to instantiate an instance, we can simply call our method.

The **@staticmethod** decorator was used to tell Python that this method is a static method.

We may use a static method to add two numbers, or print a given string.

Example:

**class** DecoratorExample:  
 """ Example Class """

**def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)

@staticmethod  
 **def** example\_function():  
 """ This method is a static method! """  
 print(**'I\'m a static method!'**)  
  
de = DecoratorExample.example\_function()

OUTPUT:

I'm a static method!

**3. Class Methods**:

Class Methods know about their class. They can’t access specific instance data, but they can call other static methods.

Class Methods do not need **self** as an argument, but they do need a parameter called **cls**. This stands for **class**, and like self, gets automatically passed in Python.

Class Methods are created using **@classmethod** decorator.

Class Methods can manipulate the class itself, which is useful when we are working larger, more complex projects.

Example:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 @classmethod  
 **def** example\_function(cls):  
 """ This method is a class method! """  
 print(**'I\'m a class method!'**)  
 cls.some\_other\_function()  
  
 @staticmethod  
 **def** some\_other\_function():  
 print(**'Hello!'**)  
  
de = DecoratorExample()  
de.example\_function()

OUTPUT:

Hello, World!

I'm a class method!

Hello!

**Summary**:

* **Instance Methods**: The most common method type. Able to access data and properties unique to each instance.
* **Static Methods**: Cannot access anything else in the class. Totally self-contained code.
* **Class Methods**: Can access limited methods in the class. Can modify class specific details.