**Topics to be covered**

* Lambda function-First class, higher order and proxy function-Iterators
* Performance of Generators: importance and needs of Generators
* Decorators: function with parameter and arguments-Class method and static method decorators
* Map, Filter, Zip, Reduce,
* Regular Expressions
* Comprehensions: List, tuples, set and Dictionary
* Modules and packages: import packages, built-in modules, package index, pip install, virtual Environments

**Functional Programming**

What Is Functional Programming?

Functional programming (FP) is a software development approach centered around the use of pure functions for creating maintainable software.

FP involves building programs through the application and composition of functions.

**It leverages language features by treating functions as variables, arguments, and return values, resulting in cleaner and more elegant code.**

Immutable data is emphasized, and shared states are avoided, distinguishing it from object-oriented programming (OOP) which often uses mutable data and shared states.

Functional programming languages prioritize declarations and expressions over statement execution.

Functions are treated as first-class citizens, allowing them to be passed as arguments, returned from other functions, and assigned to names.

FP emphasizes focusing on results rather than the process, and does not typically support iterations like loop statements or conditional statements such as If-Else.

Lambda calculus, created by Alonzo Church, studies computations with functions.

It defines computability and is as powerful as Turing machines.

It forms the theoretical basis for modern functional programming languages.

**Functional programming in Python**

Functional programming in Python involves writing code in a way that emphasizes the use of functions as the primary building blocks of programs. Here's how it's typically approached:

**Pure Functions**: Functions in functional programming should ideally be pure, meaning they have no side effects and produce the same output for the same input every time they're called.

**Immutable Data**: Data structures are preferably immutable, meaning they cannot be changed after they're created. This encourages a more declarative style of programming.

**Higher-order Functions**: Functions can be treated as first-class citizens, meaning they can be passed as arguments to other functions, returned from functions, and assigned to variables.

**Recursion**: Instead of using iterative constructs like loops, functional programming often utilizes recursion to perform repetitive tasks.

**List Comprehensions and Functional Constructs**: Python provides features like list comprehensions, generator expressions, map(), filter(), and reduce() functions, which are commonly used in functional programming paradigms.

**Avoiding Mutable State**: Functional programming discourages the use of mutable state and encourages immutable objects and data transformations.

**Lazy Evaluation**: Lazy evaluation techniques, like generators, allow for more efficient memory usage by computing values only when needed.

**lambda function**

Small anonymous function (Function without name)

Take any number of arguments, but can only have one expression

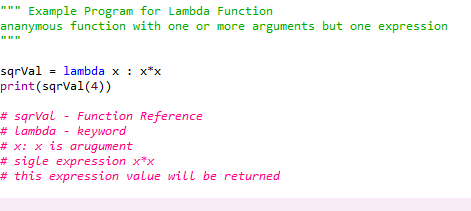
**Syntax**

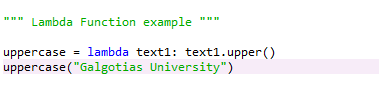
lambda arguments : Expression

**Example**

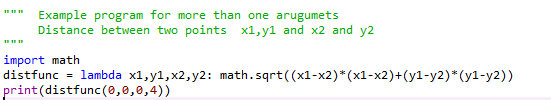
funct = lambda a : a+10

print(funct(4))



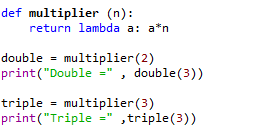


Example Lambda function with more than one arguments



## Why Use Lambda Functions?

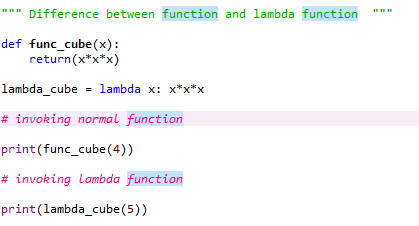
The power of lambda is better shown when you use them as an anonymous function inside another function



**double(3) executes function (lamba a:a\*2) where argument a = 3**

**Multiplier(2) = returns function (lamba a:a\*2) and saved in variable double**

### Difference Between Lambda functions and def defined function



| **With lambda function** | **Without lambda function** |
| --- | --- |
| Supports single-line sometimes statements that return some value. | Supports any number of lines inside a function block |
| Good for performing short operations/data manipulations. | Good for any cases that require multiple lines of code. |
| Using the lambda function can sometime reduce the readability of code. | We can use comments and function descriptions for easy readability. |

# First Class functions in Python

A programming language is said to support first-class functions if it treats functions as first-class objects(**First class** objects in a language are handled uniformly throughout. They may be stored in data structures, passed as arguments, or used in control structures.). Python supports the concept of First Class functions.

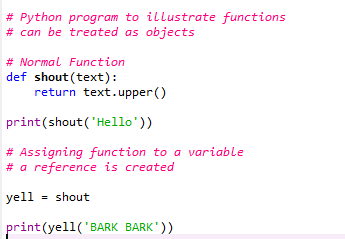
**Properties of First class functions:**

* A function is an instance of the Object type.
* You can store the function in a variable.
* You can pass the function as a parameter to another function.
* You can return the function from a function.
* You can store them in data structures such as hash tables, lists, …

**Functions as Objects**

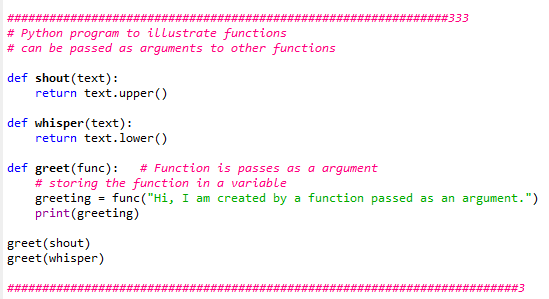
In Python, a function can be assigned to a variable.

This assignment does not call the function, instead a reference to that function is created.

****

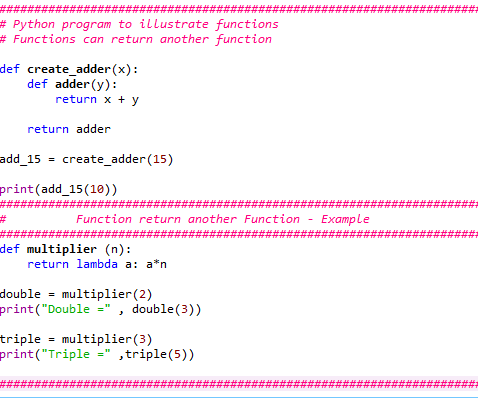
#### Passing Function as an argument to other function

Functions are like objects in Python, therefore, they can be passed as argument to other functions

****

#### Returning function

As functions are objects, we can also return a function from another function.

****

**Higher Order Functions in Python**

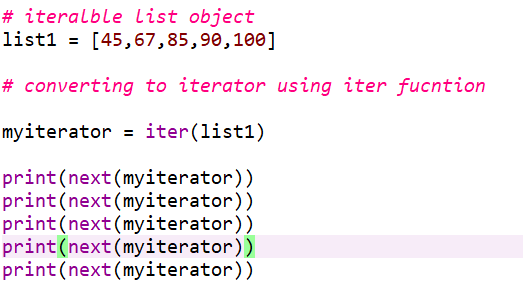
A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output

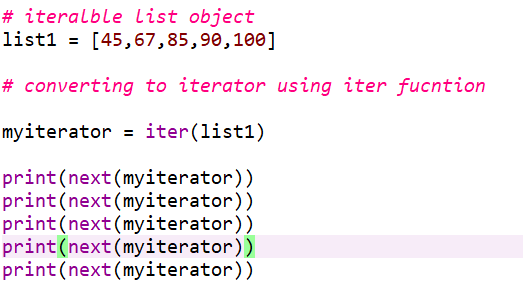
Functions that operate with another function are known as Higher order Functions

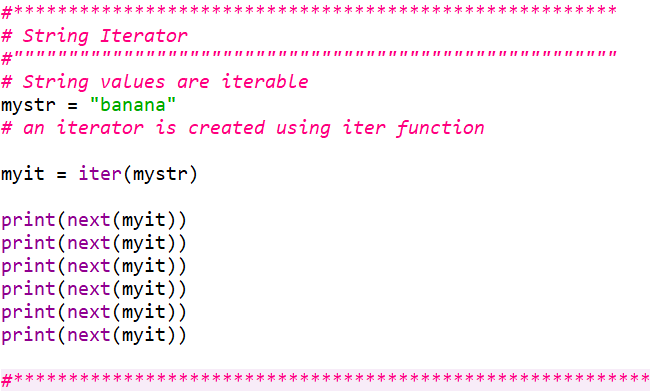
A function that receives another function as an argument or that returns a new function or both is called Higher-order function. Higher-order functions are only possible because of the First-class function.

**Iterator**

* Iterators are objects that can be iterated upon (can traverse through all the values)
* An iterator is an object which implements the iterator protocol, which consist of the methods \_\_iter\_\_() and \_\_next\_\_()
* Iterator vs Iterable
  + Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.
  + All these objects have a iter() method which is used to get an iterator**:**

****

****

****

## Create an Iterator

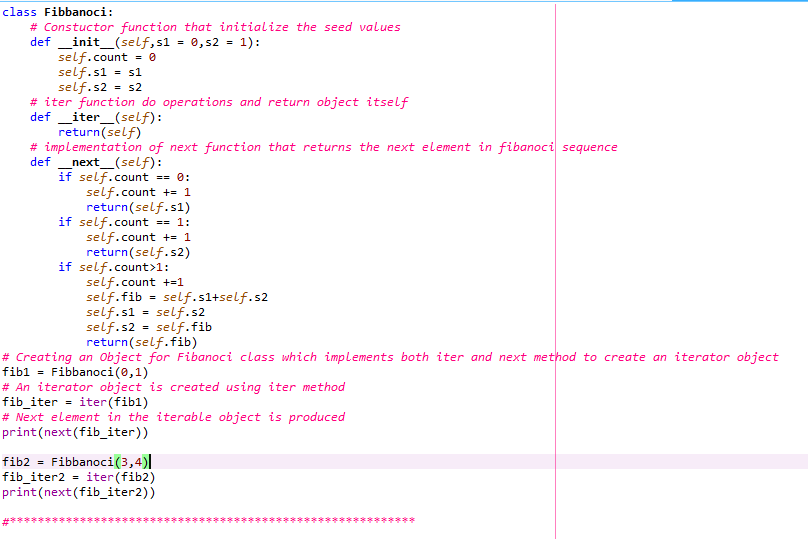
To create an object/class as an iterator you have to implement the methods \_\_iter\_\_() and \_\_next\_\_() to your object.

As you have learned in the [Python Classes/Objects](https://www.w3schools.com/python/python_classes.asp) chapter, all classes have a function called \_\_init\_\_(), which allows you to do some initializing when the object is being created.

The \_\_iter\_\_() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

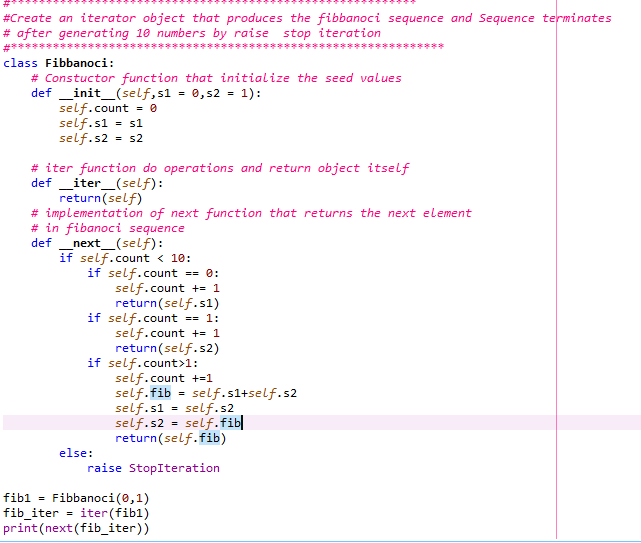
The \_\_next\_\_() method also allows you to do operations, and must return the next item in the sequence.

**Create an iterator object that produces the fibbanoci sequence**

****

To prevent the iteration from going on forever, we can use the StopIteration statement.

In the \_\_next\_\_() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

****

## Python Generators

The most expedient alternative to implementing an iterator is to use a generator. Although generators may look like ordinary [**Python functions**](https://www.datacamp.com/courses/writing-functions-in-python?hl=GB), they are different. For starters, a generator object does not return items. Instead, it uses the **yield** keyword to generate items on the fly. Thus, we can say a generator is a special kind of function that leverages [**lazy evaluation**](https://en.wikipedia.org/wiki/Lazy_evaluation). Generators do not store their contents in memory as you would expect a typical iterable to do. For example, if the goal were to find all of the factors for a positive integer, we would typically implement a traditional function (learn more about [**Python Functions**](https://www.datacamp.com/tutorial/functions-python-tutorial) in this tutorial) as follows:

def factors(n):

factor\_list = []

for val in range(1, n+1):

if n % val == 0:

factor\_list.append(val)

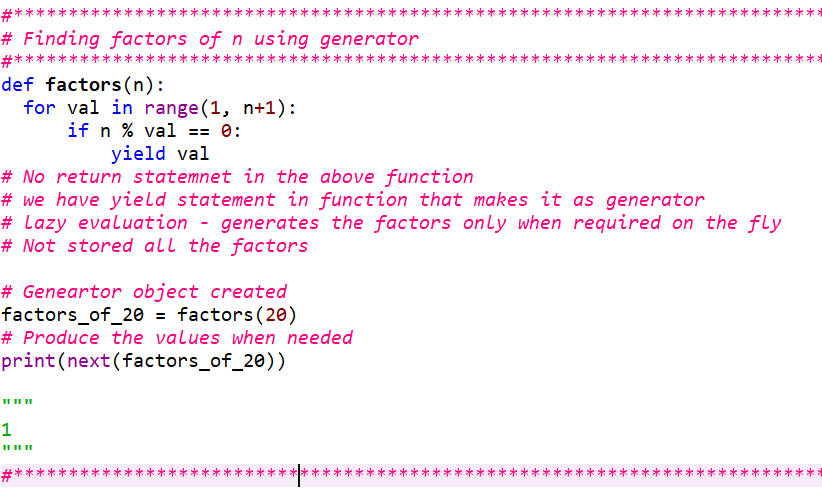
return factor\_list

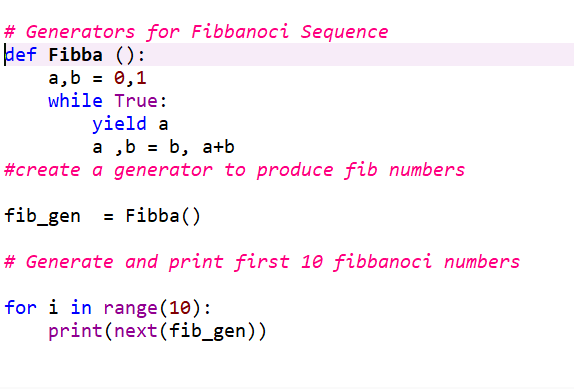
print(factors(20))

"""

[1, 2, 4, 5, 10, 20]

"""

****



Iterators Vs Generators

To recap, iterators are objects that can be iterated on, and generators are special functions that leverage lazy evaluation. Implementing your own iterator means you must create an \_\_iter\_\_() and \_\_next\_\_() method, whereas a generator can be implemented using the yield keyword in a Python function or comprehension.

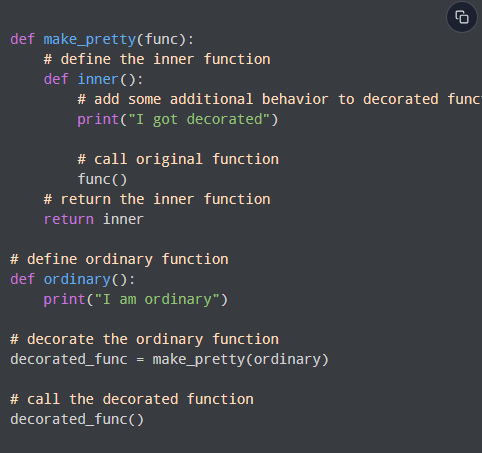
You may prefer to use a custom iterator over a generator when you require an object with complex state-maintaining behavior or if you wish to expose other methods beyond \_\_next\_\_(), \_\_iter\_\_(), and \_\_init\_\_(). On the other hand, a generator may be preferable when dealing with large sets of data since they do not store their contents in memory or when it is not necessary to implement an iterator.

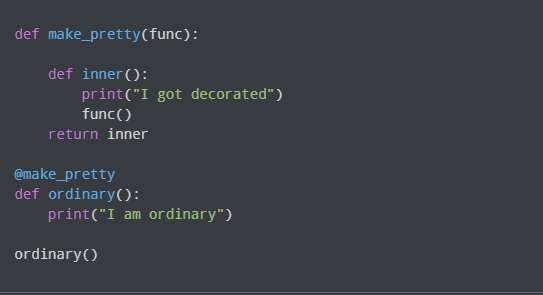
**Decorator in Python**

A Python decorator is a function that takes in a function and returns it by adding some functionality.

In fact, any object which implements the special \_\_call\_\_() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.



****

**map() function in python**

**map()** function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

**Python map() Function Syntax**

***Syntax****: map(fun, iter)*

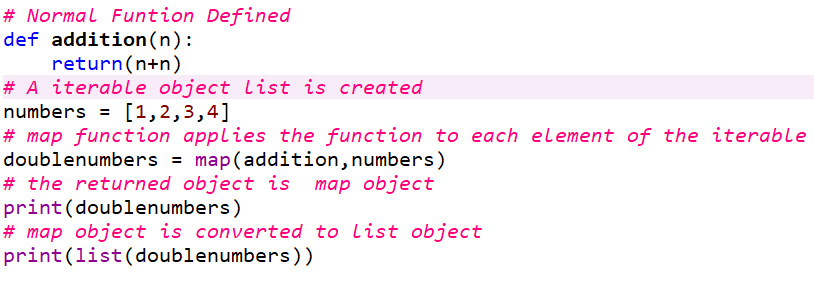
***Parameters:***

* ***fun:****It is a function to which map passes each element of given iterable.*
* ***iter:****It is iterable which is to be mapped. (list, tuple)*

***NOTE:****You can pass one or more iterable to the map() function*

***Returns:****Returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)*

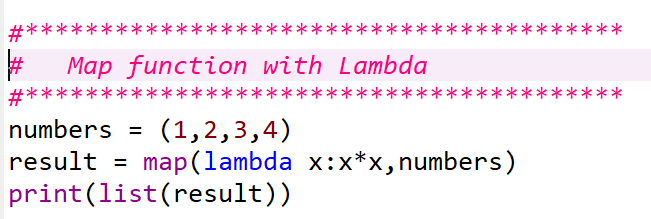
***NOTE :****The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) .*

****

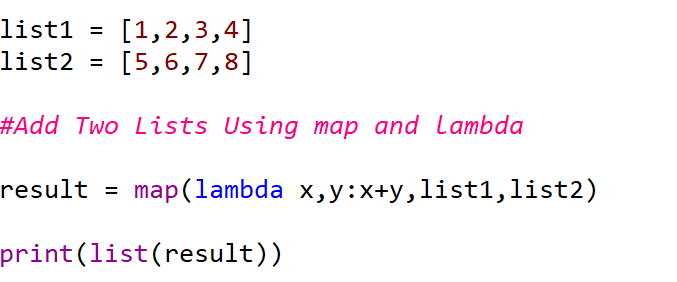
**Map() function with Lambda**

**Numbers = (1,2,3,4)**

**Map(lambda x:x+x,numbers)**

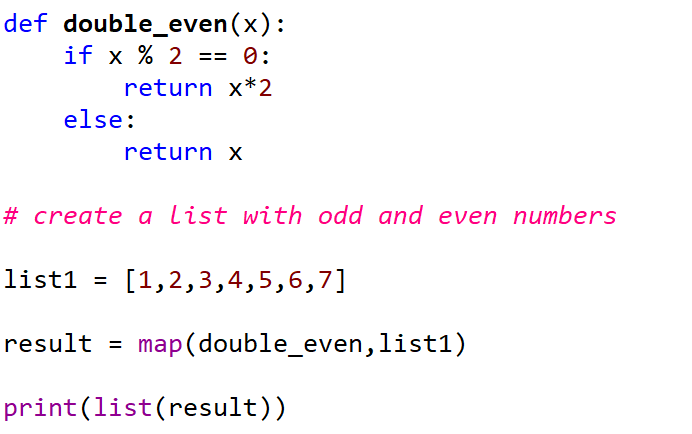
****

**map function with more than one iterable objects**

****

**Exercise**

**Write a python Program that doubles even numbers in a list and keep odd numbers as such using map function**

****

**Advantages of map()**

Utilizing `map()` offers several advantages.

map()` is implemented in C and is highly optimized,

its internal implicit loop may exhibit greater efficiency compared to a conventional Python for loop, enhancing performance.

Regarding memory consumption, employing a for loop necessitates storing the entire list in the system's memory. Conversely, with `map()`, items are retrieved on demand, ensuring that only one item occupies the system's memory at any given time. This feature can lead to more efficient memory usage when working with large datasets.

**Filter Funtion**

Filter function is used to exclude items in an iterable object. The filter() function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

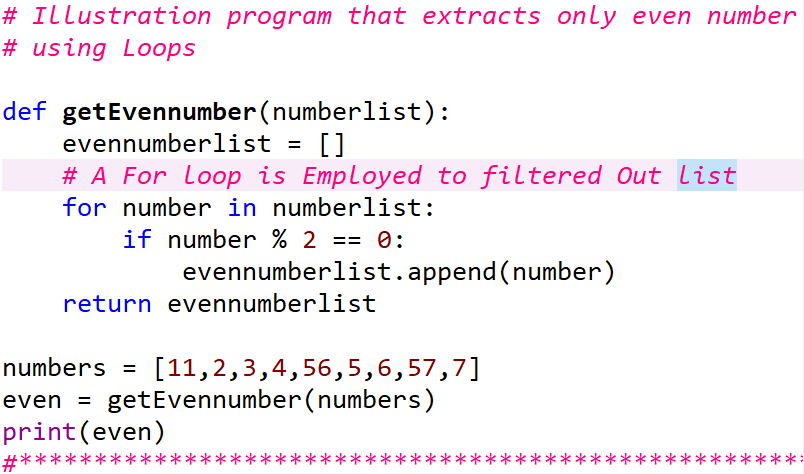
Syntax

filter(*function*, *iterable*)

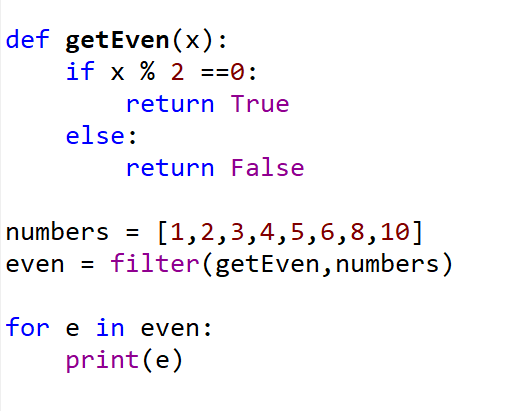
|  |  |
| --- | --- |
| *function* | A Function to be run for each item in the iterable |
| *iterable* | The iterable to be filtered |

* "function," must be a single-argument function
* The first argument for `filter()` should be a function object (meaning you pass a function without parentheses.)
* Predicate function returning True or False based on a condition
* This function acts as a decision or filtering function, determining which values to keep in the resulting iterable by filtering out those evaluated as False
* The second argument, `iterable`, accepts any Python iterable like lists, tuples, or sets, including generator and iterator objects. `filter()` exclusively accepts one iterable.
* `filter()` iterates over every item in the iterable, applying the function in a loop.
* It returns an iterator yielding values for which the function returns true.
* Importantly, this process doesn't alter the original iterable.
* As `filter()` is optimized in C, its internal loop can be more efficient than a regular for loop, enhancing execution time. This efficiency is a key advantage of using `filter()` in Python.
* Moreover, `filter()` returns a filter object, which is an iterator providing values on demand, supporting a lazy evaluation approach. This iterator-based mechanism enhances memory efficiency compared to an equivalent for loop.

**Program that filters even numbers using loops**

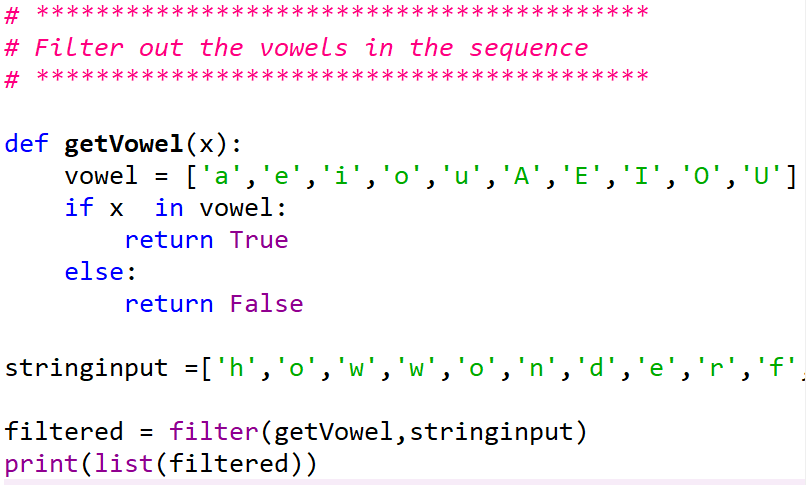


The same results can be obtained using filter function without any for loops.



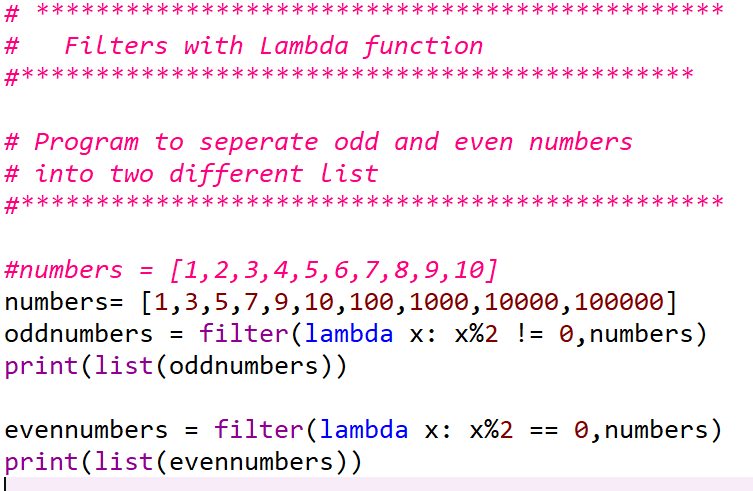
Exercise

Write a python program using filter function that extracts only the vowel characters from the stream of character inputs



Filter funtions will be normally paired with lambda function

Use the lambda function to separate list of numbers into the odd and even numbers ( two separate list )



Exercise : filter out the details of adult customers only from the list.

Sample List input : custList = [ [A,34],[B,12],[C,18],[D,10]]

# Seperating Adult Customers - Programs

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CustList = [["Anu",10],["Beena",18],["Citra",17],["Divya",45]]

def adultFilter(X):

if X[1] >=18 :

return True

else:

return False

adultlist = filter(adultFilter, CustList)

print(list(adultlist))

# Using Lambda function as predicate function

adultlist1 = filter(lambda X : X[1] >= 18,CustList)

print(list(adultlist1))

# reduce() in Python

The functools module in Python defines the reduce() method, which only delivers a single value as output after reducing the entire iterable to a single integer, string, or boolean. It does not return any iterators or multiple values.

Python's reduce() function applies a pre-defined function to each element of an iterable (such as a list, tuple, dictionary, etc.) and generates a single-valued result.

Applying the reduce function to the iterable passed as an argument yields this single-valued output, which only returns a single integer, string, or boolean.

Reduce() is a built-in function written in the C programming language, making it much more optimized and faster.

It is similar to a for loop in Python.

The functools module, which must be imported before calling the function in our program, contains the reduce() function in Python. This single value output denotes that only one integer, string, or boolean is returned due to applying the reduce function to the iterable. Let us look at the syntax of reduce function in Python.

**Syntax**

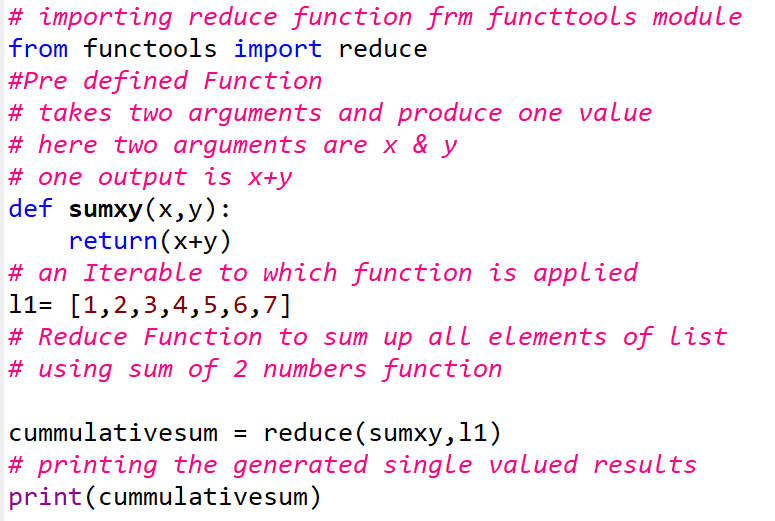
*functools.reduce(function, iterable)*

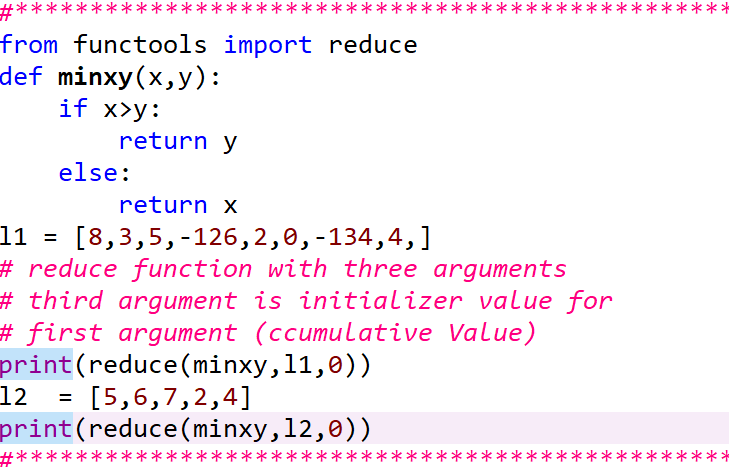
These two parameters in reduce function are

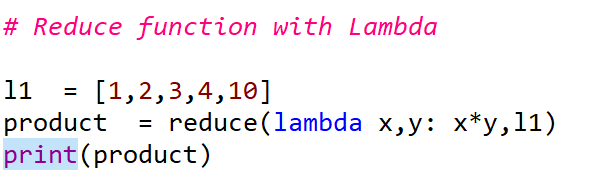
* Reduce() takes a function as its first argument. The result will be computed cumulatively, applying this function to each element in an iterable.
* The second argument can be repeated. Python objects that can be iterated over or looped over are known as iterables. Examples include lists, tuples, sets, dictionaries, generators, iterators, etc.

## Working of Reduce Function in Python

1. The iterable's first two elements are subject to the function passed as an argument.
2. The function is then applied to both the most recent result that was generated and the subsequent element in the iterable.
3. The iterable is processed until the end of this process.
4. Applying the reduce function to the iterable causes the single value to be returned.



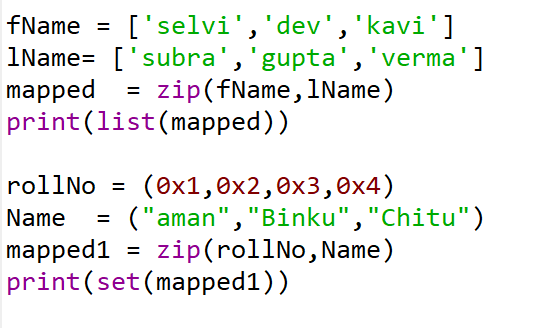


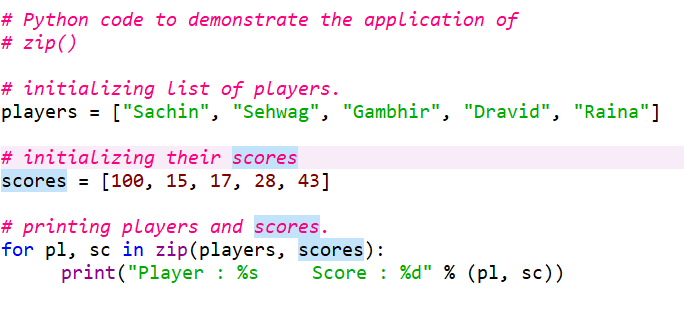


**Zip Function in Python**

In functional programming in Python, the `zip()` function is often used to combine multiple iterables element-wise into tuples. It takes two or more iterables as input and returns an iterator of tuples, where each tuple contains the corresponding elements from the input iterables.

In functional programming, `zip()` is commonly used with other functional constructs like `map()` or list comprehensions to perform operations onmultiple iterables simultaneously. This allows for a more concise and expressive way to manipulate data in a functional style.





**Pure Functions**

A pure function adheres strictly to the principle that its output is solely determined by its input, devoid of any observable side effects. In functional programming, programs consist entirely of the evaluation of pure functions. Computation progresses through nested or composed function calls, without altering state or mutable data.

The functional paradigm enjoys popularity due to its numerous advantages over other programming paradigms. Functional code exhibits the following characteristics:

1. High-level: Descriptions of desired results take precedence over explicit steps needed to achieve them. Single statements are often succinct yet impactful.

2. Transparent: The behavior of a pure function relies solely on its inputs and outputs, excluding any intermediary values. This absence of side effects simplifies debugging.

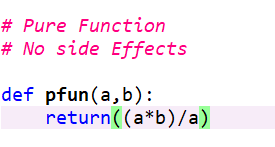
3. Parallelizable: Functions devoid of side effects can be executed more readily in parallel with each other, enhancing efficiency in concurrent operations.

To support functional programming, it's advantageous for a programming language to enable functions to:

1. Accept other functions as arguments.

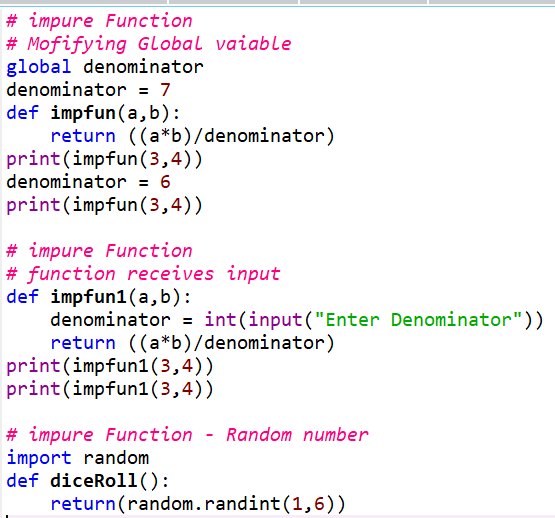
2. Return functions to their caller.

Python excels in both these regards. As covered previously, Python treats everything in a program as an object, with functions being no exception.



**Impure Functions**

* An impure function is a function that has side effects or does not always return the same output when given the same input. Side effects can include
  + Modifying a global variable
  + Changing the state of an object
  + Functions that generate Random output
  + Functions receive user inputs

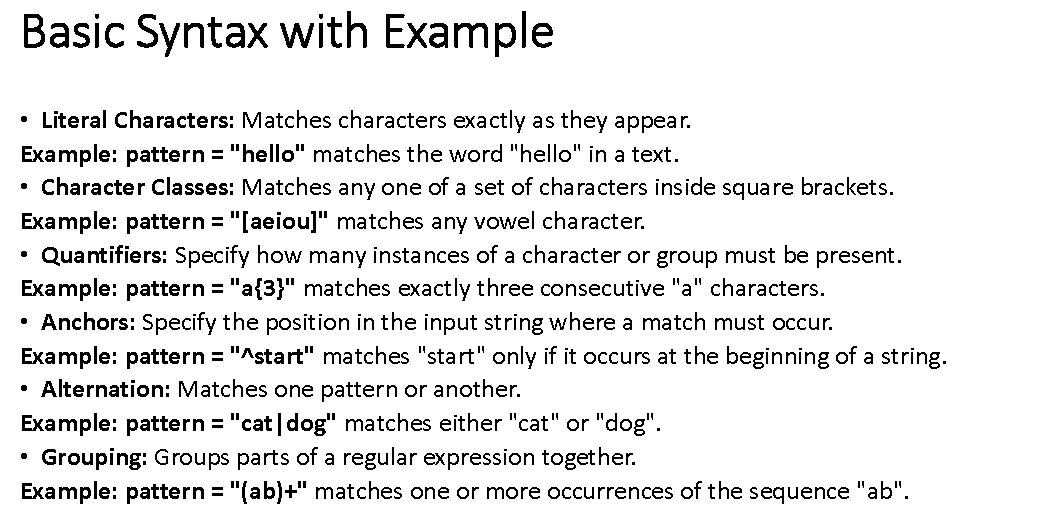
****

Regular Expression

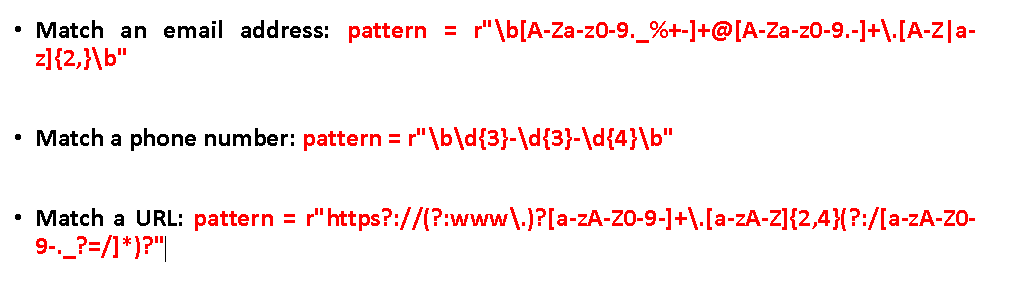
A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

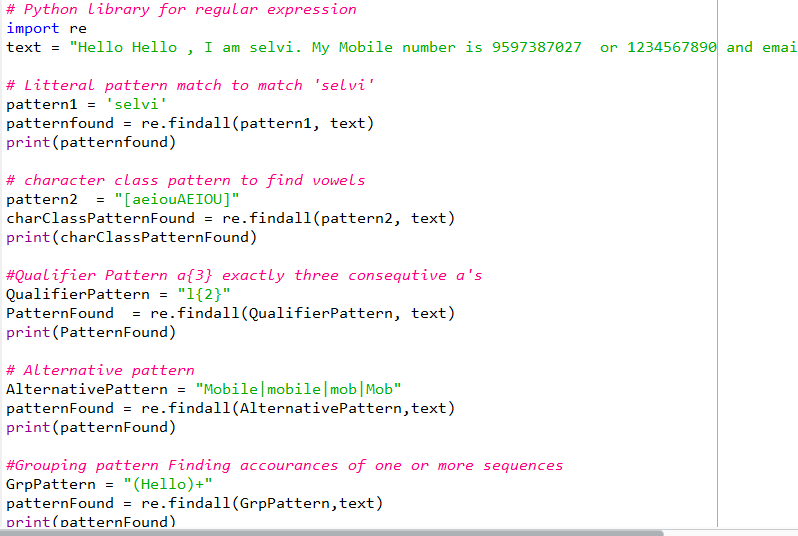
RegEx can be used to check if a string contains the specified search pattern.

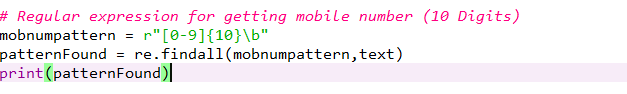
[Python](https://www.geeksforgeeks.org/python-programming-language/)has a built-in module named “**re”**that is used for regular expressions in Python. We can import this module by using the [import statement](https://www.geeksforgeeks.org/import-module-python/).



Example matches







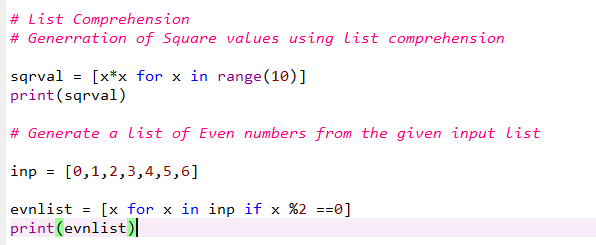
# Comprehensions in Python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using previously defined sequences. [Python](https://www.geeksforgeeks.org/python-programming-language/) supports the following 4 types of comprehension:

* List Comprehensions
* Dictionary Comprehensions
* Set Comprehensions
* Generator Comprehensions

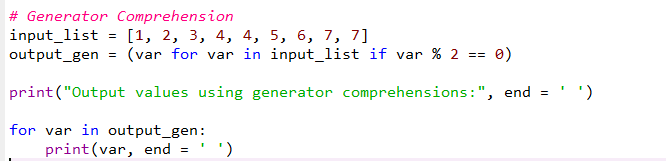
List comprehensions provide a concise way to create lists in Python by applying an expression to each item in an iterable.

# Generating a list of Sqr values using list comprehension



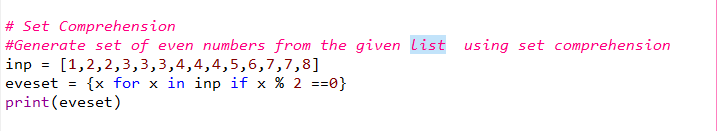
## Generator Comprehensions

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don’t allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient.



## Set Comprehensions

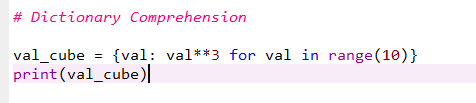
Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }



## Dictionary Comprehensions

Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

*output\_dict = {key:value for (key, value) in*iterable *if (key, value satisfy this condition)}*

**

Packages, Modules, installing packages, Virtual environment …. Refer the presentation PDF file