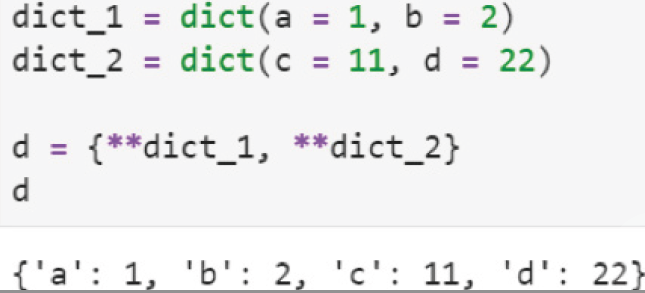
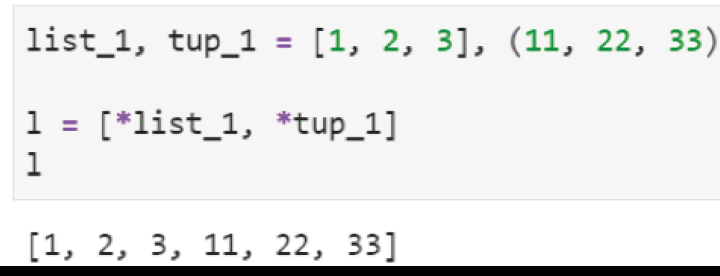
* An interpreter translates high level code and executes it one line at a time. This makes interpreted languages highly interactive due to the immediate feedback and A compiler converts complete/whole high level programs, into low level code called the object code, that can be executed as and when required.
* Sets can only contain immutable objects.

Iterators, generators:

* Iter() and next() are the built in functions which are used to create the iterator and get the next value of the iterator respectively.
* For loop is executed using iter() and next() functions.
* Once the iterator reaches end stopiteration error is raised.

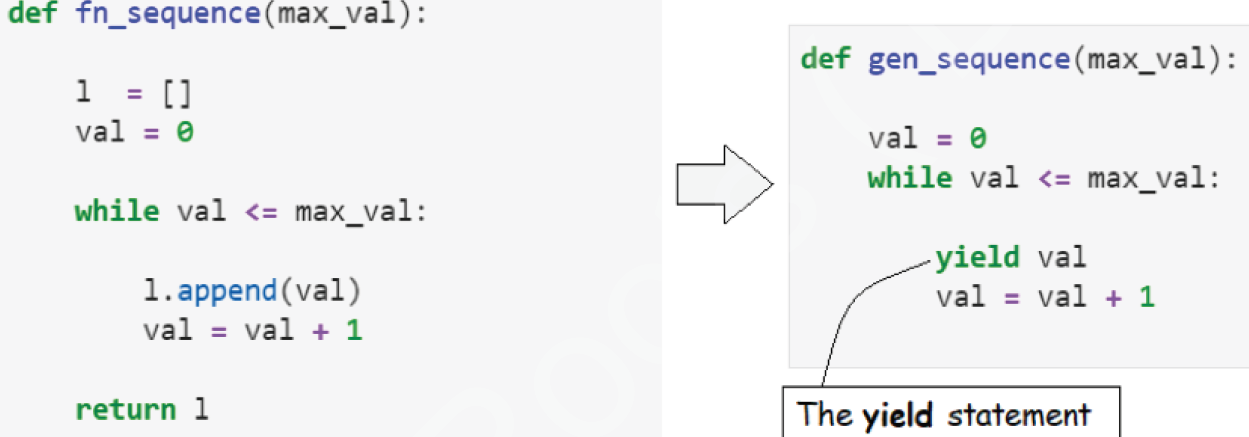
**UNPACKING CONTAINERS USING THE STAR & DOUBLE STAR OPERATORS:**

* The **star** symbol apart from being used to represent the multiplication operator, can be used to unpack iterable sequence containers (ie: lists & tuples) into other sequence containers.
* The double star operator can be used to unpack dictionaries into any other dictionary.

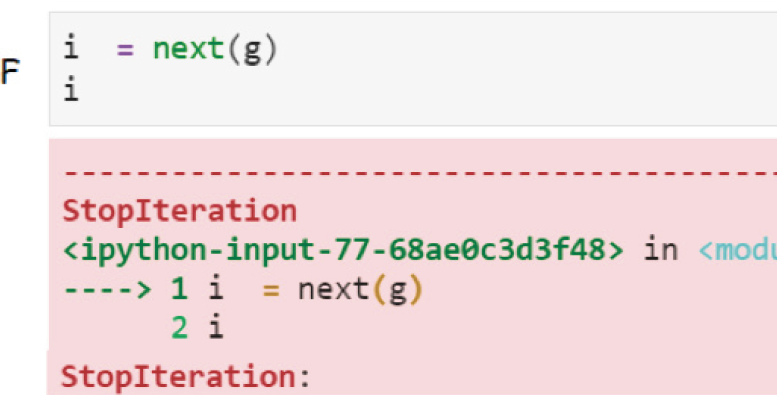
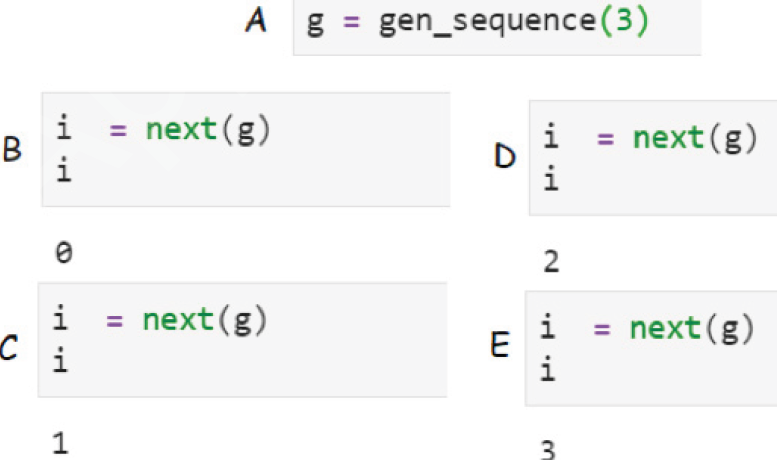


**Generator functions:**

* Generator functions allow us to create custom iterators.



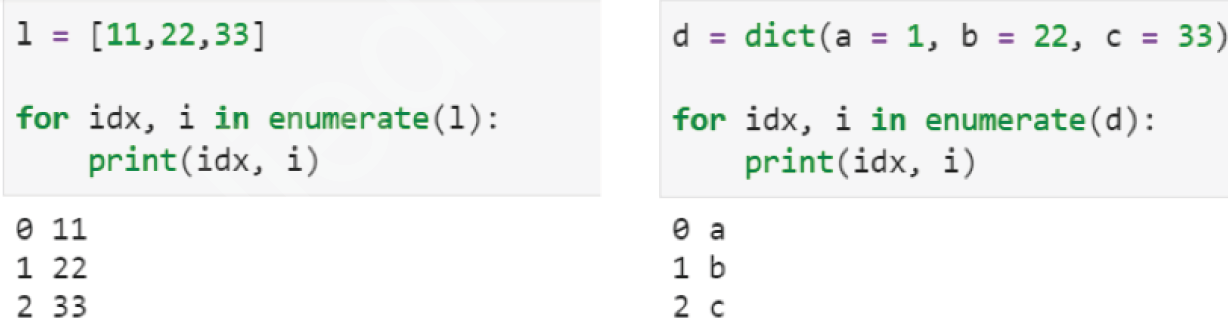
* The code construct on the right side of the image above is called a **generator function**.
* Unlike the normal function on the left, which **returns** a list of values when **called**, the generator on the right **yields** one value at a time, each time the **next** function is applied to it.



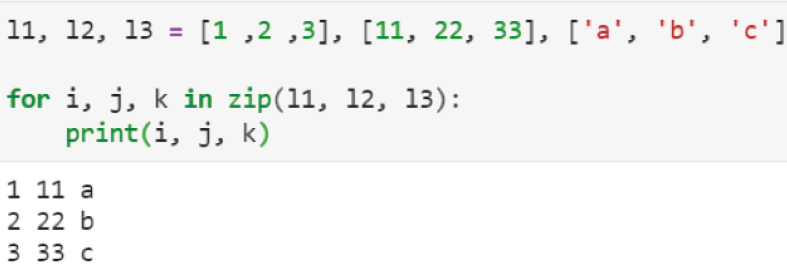
* The main difference between functions and generators is that generators do not use the return statement, they use the yield statement instead.
* When we call a generator function as shown below, python returns a generator object and this generator object is assigned to the variable name specified
* When the **next** function is applied repetitively to the generator object assigned to variable name **g**, the following things happen:
* When the next function is applied to g for the first time, the code within gen\_sequence gets executed till it reaches the yield statement. When the yield statement is executed, it returns the value specified after the yield keyword and pauses the execution flow at that point
* The next time the next function is called on the generator object g, the code resumes execution from where it was stopped. In other words, any remaining code after the yield statement is executed and the while loop resumes till the yield statement is encountered again. At this point the value specified after the yield keyword is returned and the execution flow paused again.
* Step 2 can be repeated until python raises the StopIteration error, when the condition specified after the while keyword no longer holds true.

**INBUILT GENERATORS/ITERATORS:**

* Enumerate:
* The enumerate iterator function takes in an iterable object (list/tuple/dict) and returns tuples containing the value being currently yielded and its count or index value as shown below



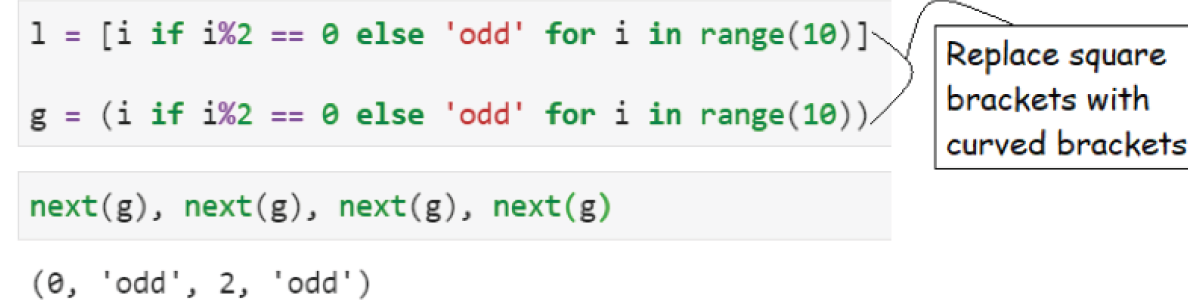
ZIP: The zip iterator allows us to iterate through multiple equi-sized containers simultaneously.



RANGE():  
The range iterator function is an iterator that yields sequences based on the **start, end** and **step size** values fed to it.

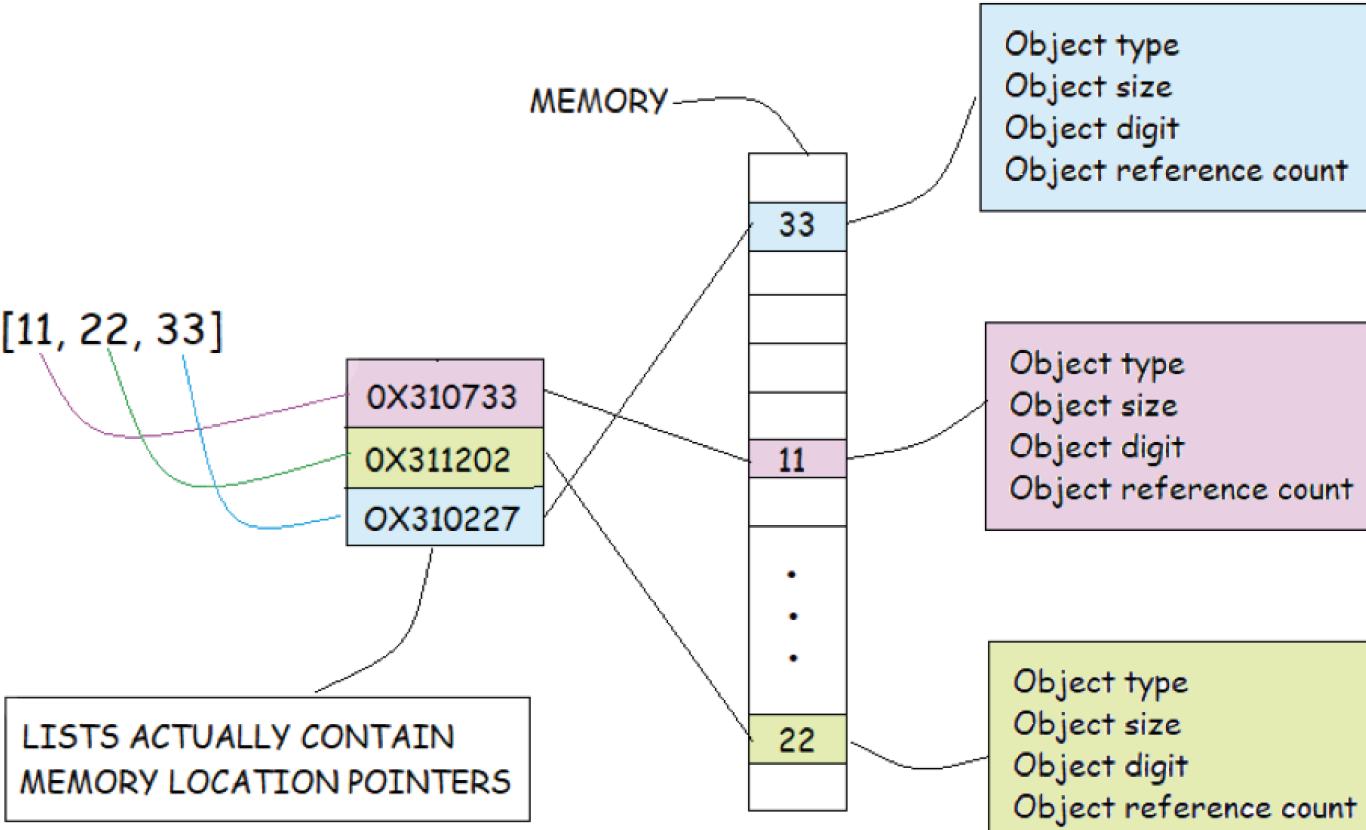
**GENERATOR COMPREHENSIONS:**

* Simple generators can be created by converting list comprehensions to generator comprehensions as shown below:



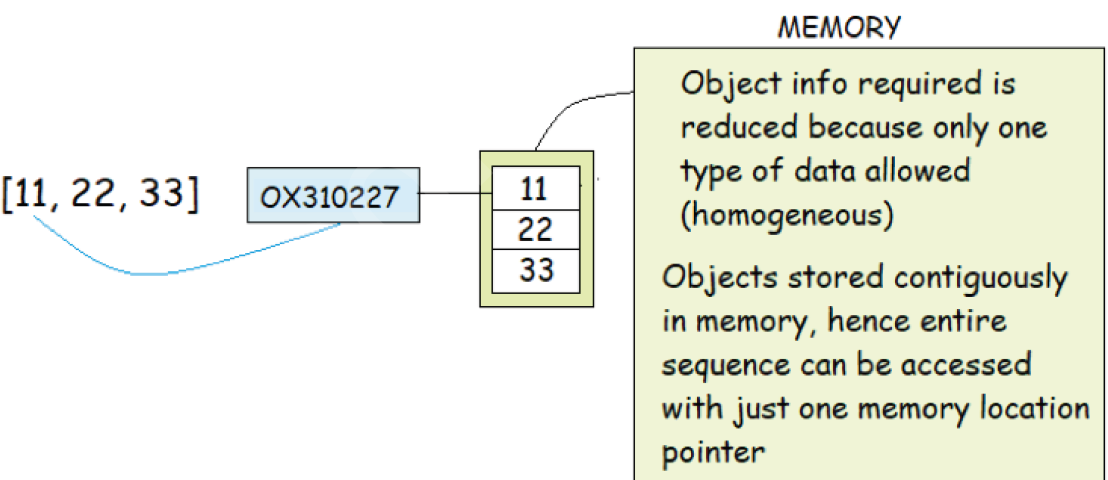
**LISTS Vs ARRAYS:**

* lists are data container objects that are ordered, mutable and heterogeneous.
* The flexibility of lists comes at a price. They are not efficient in terms of numerical processing. Lists actually do not contain any object, but instead they contain mappings/pointers to objects that are stored at various locations within the memory of the computer.



* Due to the **non-contiguous** nature of how objects belonging to a list are stored within the memory, they are not fast to iterate/compute through. Also, due to the amount of information each of those objects contain (as shown above), they tend to use up more memory. In other words because the objects grouped within a list are stored in various locations in the memory, python has to jump around the memory space each time it has to retrieve an object within it. The situation becomes compounded when we consider lists that contain lists or any other container object within them, instead of just integers and floats.

Numpy nd array :



* Structuring a data container object in the way described above makes it less flexible, but at the same time the values within the resulting data container become easily accessible, thus making computation faster and also since lesser information is required to define the container object, it requires lesser memory space.
* Numpy’s ndarray data container object, due to its “type” constraint and contiguous memory storage design, enables one to perform vectorized operations (ie: fast computations on all the objects contained within the array without needing to write loops).
* Numpy basically uses one fundamental data container type for all its purposes - the ndarray or array as it is commonly called. An array can contain a homogeneous collection of any of the value types described in the table above. This homogeneous collection of values can be ordered along more than one dimensions.

Pandas Append:

* DataFrame.append(*other*, *ignore\_index=False*, *verify\_integrity=False*, *sort=False*)
* Append rows of *other* to the end of caller, returning a new object
* Columns in *other* that are not in the caller are added as new columns.

Pandas concat :

* pandas.concat(objs, axis=0, join='outer', ignore\_index=False, keys=None, levels=None, names=None, verify\_integrity=False, sort=False, copy=True)
* Concatenate pandas objects along a particular axis with optional set logic along the other axes.
* Can also add a layer of hierarchical indexing on the concatenation axis

Parameters:

* Join{‘inner’, ‘outer’}, default ‘outer’
* Operates on the indexs only.

Pandas join:

* DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
* Join columns of another DataFrame.
* Join columns with other DataFrame **either on index or on a key column**. Efficiently join multiple DataFrame objects **by index** at once by passing a list.

Parameters:

* how{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘left’

pandas merge:

* pandas.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=False, suffixes=('\_x', '\_y'), copy=True, indicator=False, validate=None)
* Merge DataFrame or named Series objects with a database-style join
* The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

Parameters:

* how{‘left’, ‘right’, ‘outer’, ‘inner’, ‘cross’}, default ‘inner’

main differences between merge an join:

* The main difference between join vs merge would be; join() is used to combine two DataFrames on the index but not on columns whereas merge() is primarily used to specify the columns you wanted to join on, this also supports joining on indexes and combination of index and columns.
* Both these methods support left on the column and right on the index however, merge additionally supports left on the index and right on the column.

