F# - SEQUENCES

Sequences, like lists also represent an ordered collection of values. However, the elements in a sequence or sequence expression are computed when required. They are not computed at once, and for this reason they are used to represent infinite data structures.

Defining Sequences

Sequences are defined using the following syntax −

```
seq { expr }
```

For example,

```
let seq1 = seq { 1 .. 10 }
```

Creating Sequences and Sequences Expressions

Similar to lists, you can create sequences using ranges and comprehensions.

Sequence expressions are the expressions you can write for creating sequences. These can be done −

- By specifying the range.
- By specifying the range with increment or decrement.
- By using the **yield** keyword to produce values that become part of the sequence.
- By using the → operator.

The following examples demonstrate the concept −

Example 1

```
(* Sequences *)
let seq1 = seq { 1 .. 10 }

(* ascending order and increment*)
printfn "The Sequence: %A" seq1
let seq2 = seq { 1 .. 5 .. 50 }

(* descending order and decrement*)
printfn "The Sequence: %A" seq2
let seq3 = seq {50 .. -5 .. 0}
printfn "The Sequence: %A" seq3

(* using yield *)
let seq4 = seq { for a in 1 .. 10 do yield a, a*a, a*a*a }
printfn "The Sequence: %A" seq4
```

When you compile and execute the program, it yields the following output −

```
The Sequence: seq [1; 2; 3; 4; ...]
The Sequence: seq [1; 6; 11; 16; ...]
The Sequence: seq [50; 45; 40; 35; ...]
The Sequence: seq [(1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64); ...]
```

Example 2

The following program prints the prime numbers from 1 to 50 −

```
(* Recursive isprime function. *)
let isprime n =
   let rec check i =
      i > n/2 || (n % i <> 0 && check (i + 1))
   check 2

let primeIn50 = seq { for n in 1..50 do if isprime n then yield n }
for x in primeIn50 do
   printfn "%d" x
```

When you compile and execute the program, it yields the following output −

```
1
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

Basic Operations on Sequence

The following table shows the basic operations on sequence data type −

Value	Description
append : seq<'T> → seq<'T> → seq<'T>	Wraps the two given enumerations as a single concatenated enumeration.
average : seq<^T> → ^T	Returns the average of the elements in the sequence.
averageBy : $'T \rightarrow {}^{U}$ → seq<'T> → ^U	Returns the average of the results generated by applying the function to each element of the sequence.
cache : seq<'T> → seq<'T>	Returns a sequence that corresponds to a cached version of the input sequence.
cast : IEnumerable → seq<'T>	Wraps a loosely-typed System. Collections sequence as a typed sequence.
choose : $'T \rightarrow 'Uoption$ → seq<'T> → seq<'U>	Applies the given function to each element of the list. Return the list comprised of the results for each element where the function returns **Some**.
collect : $'T \rightarrow 'Collection$ → seq<'T> → seq<'U>	Applies the given function to each element of the sequence and concatenates all the results.
compareWith : $'T \rightarrow 'T \rightarrow int$ → seq<'T> → seq<'T> → int	Compares two sequences using the given comparison function, element by element.
concat : seq<'Collection> → seq<'T>	Combines the given enumeration-of-enumerations as a single concatenated enumeration.

countBy : 'T → 'Key → seq<'T> → seq<'Key * int>	Applies a key-generating function to each element of a sequence and return a sequence yielding unique keys and their number of occurrences in the original sequence.
delay : unit → seq <'T > → seq<'T>	Returns a sequence that is built from the given delayed specification of a sequence.
distinct : seq<'T> → seq<'T>	Returns a sequence that contains no duplicate entries according to generic hash and equality comparisons on the entries. If an element occurs multiple times in the sequence then the later occurrences are discarded.
distinctBy : 'T → 'Key → seq<'T> → seq<'T>	Returns a sequence that contains no duplicate entries according to the generic hash and equality comparisons on the keys returned by the given key-generating function. If an element occurs multiple times in the sequence then the later occurrences are discarded.
empty : seq<'T>	Creates an empty sequence.
exactlyOne : seq<'T> → 'T	Returns the only element of the sequence.
exists : 'T → bool → seq<'T> → bool	Tests if any element of the sequence satisfies the given predicate.
exists2 : 'T1 → 'T2 → bool → seq<'T1> → seq<'T2> → bool	Tests if any pair of corresponding elements of the input sequences satisfies the given predicate.
filter : 'T → bool → seq<'T> → seq<'T>	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**.
find : 'T → bool → seq<'T> → 'T	Returns the first element for which the given function returns **true**.
findIndex : 'T → bool → seq<'T> → int	Returns the index of the first element for which the given function returns **true**.
fold : 'State → 'T → 'State → 'State → seq<'T> → 'State	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, then this function computes f . . . (fsi0...) iN.
forall : 'T → bool → seq<'T> → bool	Tests if all elements of the sequence satisfy the given predicate.
forall2 : 'T1 → 'T2 → bool → seq<'T1> → seq<'T2> → bool	Tests the all pairs of elements drawn from the two sequences satisfy the given predicate. If one sequence is shorter than the other then the remaining elements of the longer sequence are ignored.
groupBy : 'T → 'Key → seq<'T> → seq<'Key * seq<'T>>	Applies a key-generating function to each element of a sequence and yields a sequence of unique keys. Each unique key has also contains a sequence of all elements that match to this key.
head : seq<'T> → 'T	Returns the first element of the sequence.
init : int → int → 'T → seq<'T>	Generates a new sequence which, when iterated, returns successive elements by calling the given

	function, up to the given count. The results of calling the function are not saved, that is, the function is reapplied as necessary to regenerate the elements. The function is passed the index of the item being generated.
initInfinite : *int* → *'T* → seq<'T>	Generates a new sequence which, when iterated, will return successive elements by calling the given function. The results of calling the function are not saved, that is, the function will be reapplied as necessary to regenerate the elements. The function is passed the index of the item being generated.
isEmpty : seq<'T> → bool	Tests whether a sequence has any elements.
iter : *'T* → *unit* → seq<'T> → unit	Applies the given function to each element of the collection.
iter2 : *'T1* → *'T2* → *unit* → seq<'T1> → seq<'T2> → unit	Applies the given function to two collections simultaneously. If one sequence is shorter than the other then the remaining elements of the longer sequence are ignored.
iteri : *int* → *'T* → *unit* → seq<'T> → unit	Applies the given function to each element of the collection. The integer passed to the function indicates the index of element.
last : seq<'T> → 'T	Returns the last element of the sequence.
length : seq<'T> → int	Returns the length of the sequence.
map : *'T* → *'U* → seq<'T> → seq<'U>	Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection. The given function will be applied as elements are demanded using the MoveNext method on enumerators retrieved from the object.
map2 : *'T1* → *'T2* → *'U* → seq<'T1> → seq<'T2> → seq<'U>	Creates a new collection whose elements are the results of applying the given function to the corresponding pairs of elements from the two sequences. If one input sequence is shorter than the other then the remaining elements of the longer sequence are ignored.
mapi : *int* → *'T* → *'U* → seq<'T> → seq<'U>	Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection. The integer index passed to the function indicates the index *from*0 of element being transformed.
max : seq<'T> → 'T	Returns the greatest of all elements of the sequence, compared by using Operators.max.
maxBy : *'T* → *'U* → seq<'T> → 'T	Returns the greatest of all elements of the sequence, compared by using Operators.max on the function result.
min : seq<'T> → 'T	Returns the lowest of all elements of the sequence, compared by using Operators.min.
minBy : *'T* → *'U* → seq<'T> → 'T	Returns the lowest of all elements of the sequence, compared by using Operators.min on the function result.
nth : int → seq<'T> → 'T	Computes the *nth* element in the collection.

ofArray : 'T array → seq<'T>	Views the given array as a sequence.
ofList : 'T list → seq<'T>	Views the given list as a sequence.
pairwise : seq<'T> → seq<'T * 'T>	Returns a sequence of each element in the input sequence and its predecessor, with the exception of the first element which is only returned as the predecessor of the second element.
pick : $'T \rightarrow 'U option$ → seq<'T> → 'U	Applies the given function to successive elements, returning the first value where the function returns a **Some** value.
readonly : seq<'T> → seq<'T>	Creates a new sequence object that delegates to the given sequence object. This ensures the original sequence cannot be rediscovered and mutated by a type cast. For example, if given an array the returned sequence will return the elements of the array, but you cannot cast the returned sequence object to an array.
reduce : $'T \rightarrow 'T \rightarrow 'T$ → seq<'T> → 'T	Applies a function to each element of the sequence, threading an accumulator argument through the computation. Begin by applying the function to the first two elements. Then feed this result into the function along with the third element and so on. Return the final result.
scan : $'State \rightarrow 'T \rightarrow 'State$ → 'State → seq<'T> → seq<'State>	Like Seq.fold, but computes on-demand and returns the sequence of intermediary and final results.
singleton : 'T → seq<'T>	Returns a sequence that yields one item only.
skip : int → seq<'T> → seq<'T>	Returns a sequence that skips a specified number of elements of the underlying sequence and then yields the remaining elements of the sequence.
skipWhile : $'T \rightarrow bool$ → seq<'T> → seq<'T>	Returns a sequence that, when iterated, skips elements of the underlying sequence while the given predicate returns **true,** and then yields the remaining elements of the sequence.
sort : seq<'T> → seq<'T>	Yields a sequence ordered by keys.
sortBy : $'T \rightarrow 'Key$ → seq<'T> → seq<'T>	Applies a key-generating function to each element of a sequence and yield a sequence ordered by keys. The keys are compared using generic comparison as implemented by Operators.compare.
sum : seq<^T> → ^T	Returns the sum of the elements in the sequence.
sumBy	Returns the sum of the results generated by applying the function to each element of the sequence.
take : int → seq<'T> → seq<'T>	Returns the first elements of the sequence up to a specified count.
takeWhile : $'T \rightarrow bool$ → seq<'T> → seq<'T>	Returns a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns **true,** and then returns

	no further elements.
toArray : seq<'T> → 'T[]	Creates an array from the given collection.
toList : seq<'T> → 'T list	Creates a list from the given collection.
truncate : int → seq<'T> → seq<'T>	Returns a sequence that when enumerated returns no more than a specified number of elements.
tryFind : $'T \rightarrow bool$ → seq<'T> → 'T option	Returns the first element for which the given function returns **true,** or **None** if no such element exists.
tryFindIndex : $'T \rightarrow bool$ → seq<'T> → int option	Returns the index of the first element in the sequence that satisfies the given predicate, or **None** if no such element exists.
tryPick : $'T \rightarrow 'Uoption$ → seq<'T> → 'U option	Applies the given function to successive elements, returning the first value where the function returns a **Some** value.
unfold : $'State \rightarrow 'T * 'Stateoption$ → 'State → seq<'T>	Returns a sequence that contains the elements generated by the given computation.
where : $'T \rightarrow bool$ → seq<'T> → seq<'T>	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**. A synonym for Seq.filter.
windowed : int → seq<'T> → seq<'T []>	Returns a sequence that yields sliding windows of containing elements drawn from the input sequence. Each window is returned as a fresh array.
zip : seq<'T1> → seq<'T2> → seq<'T1 * 'T2>	Combines the two sequences into a list of pairs. The two sequences need not have equal lengths − when one sequence is exhausted any remaining elements in the other sequence are ignored.
zip3 : seq<'T1> → seq<'T2> → seq<'T3> → seq<'T1 * 'T2 * 'T3>	Combines the three sequences into a list of triples. The sequences need not have equal lengths − when one sequence is exhausted any remaining elements in the other sequences are ignored.

The following examples demonstrate the uses of some of the above functionalities −

Example 1

This program creates an empty sequence and fills it up later −

```
(* Creating sequences *)
let emptySeq = Seq.empty
let seq1 = Seq.singleton 20

printfn"The singleton sequence:"
printfn "%A " seq1
printfn"The init sequence:"

let seq2 = Seq.init 5 (fun n -> n * 3)
Seq.iter (fun i -> printf "%d " i) seq2
printfn""

(* converting an array to sequence by using cast *)
```

```
printfn"The array sequence 1:"
let seq3 = [| 1 .. 10 |] :> seq<int>
Seq.iter (fun i -> printf "%d " i) seq3
printfn""

(* converting an array to sequence by using Seq.ofArray *)
printfn"The array sequence 2:"
let seq4 = [| 2..2.. 20 |] |> Seq.ofArray
Seq.iter (fun i -> printf "%d " i) seq4
printfn""
```

When you compile and execute the program, it yields the following output −

```
The singleton sequence:
seq [20]
The init sequence:
0 3 6 9 12
The array sequence 1:
1 2 3 4 5 6 7 8 9 10
The array sequence 2:
2 4 6 8 10 12 14 16 18 20
```

Please note that −

- The Seq.empty method creates an empty sequence.
- The Seq.singleton method creates a sequence of just one specified element.
- The Seq.init method creates a sequence for which the elements are created by using a given function.
- The Seq.ofArray and Seq.ofList<'T> methods create sequences from arrays and lists.
- The Seq.iter method allows iterating through a sequence.

Example 2

The Seq.unfold method generates a sequence from a computation function that takes a state and transforms it to produce each subsequent element in the sequence.

The following function produces the first 20 natural numbers −

```
let seq1 = Seq.unfold (fun state -> if (state > 20) then None else Some(state, state + 1))
0
printfn "The sequence seq1 contains numbers from 0 to 20."
for x in seq1 do printf "%d " x
printfn" "
```

When you compile and execute the program, it yields the following output −

```
The sequence seq1 contains numbers from 0 to 20.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Example 3

The Seq.truncate method creates a sequence from another sequence, but limits the sequence to a specified number of elements.

The Seq.take method creates a new sequence that contains a specified number of elements from the start of a sequence.

```
let mySeq = seq { for i in 1 .. 10 -> 3*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takeSeq = Seq.take 5 mySeq
```

```
printfn"The original sequence"
Seq.iter (fun i -> printf "%d " i) mySeq
printfn""

printfn"The truncated sequence"
Seq.iter (fun i -> printf "%d " i) truncatedSeq
printfn""

printfn"The take sequence"
Seq.iter (fun i -> printf "%d " i) takeSeq
printfn""
```

When you compile and execute the program, it yields the following output −

```
The original sequence
3 6 9 12 15 18 21 24 27 30
The truncated sequence
3 6 9 12 15
The take sequence
3 6 9 12 15
```