**Generators and Iterators**

|  |  |
| --- | --- |
| **Iterator** enables traversing a set of items, providing the next item on each iteration. The set of items could come from an array, a list, a tuple or possibly be a generated sequence.  **Generator** a special type of iterator which provides the next item without necessarily first obtaining the entire set of items |  |
| **LISP - mapcar function iterator**  (**mapcar** *funcNameLiteral* *list*)  Returns a list resulting from applying the specified function to each item in the list.  (**mapcar** *funcNameLiteral* *list1 list2*)  Returns a list resulting from applying the specified function to corresponding items in each list. If one list has fewer items than the other, it simply stops with the shorter. | (mapcar '1+ '(1 2 3))  (2 3 4)  (mapcar '+ '(1 2 3) '(4 5 6))  (5 7 9)  (mapcar '+ '(1 2 ) '(4 5 6))  (5 7)  ;;; Previously we had coded REPTOP as follows:  (defun REPTOP (match rep L)  (cond ( (NULL L) () )  ( (EQL (car L) match)  (cons rep (REPTOP match rep (cdr L)) )  )  ( T (cons (car L) (REPTOP match rep (cdr L)) ) )  )  )  > (reptop 'i 'o '(m i s s i s s i p p i))  (m o s s o s s o p p o)  ;;; How can we use MAPCAR to simplify it?  (defun REPTOP (match rep L)  (MAPCAR 'ReplaceOne L)))  (defun ReplaceOne (value)  (COND ( (EQL value match) rep)  ( T value)  ) )    > (reptop 'i 'o '(m i s s i s s i p p i))  \*\*\* - COND: variable MATCH has no value  ;;; Why did that happen? Lexical(Static) scope, wouldn’t know about match  ;;; How can we fix that problem?  ;;; approach #1 use dynamic scope  (defVar match) // makes it dynamic  (defVar rep) // same  > (reptop 'i 'o '(m i s s i s s i p p i))  (m o s s o s s o p p o)  ;;; approach #2 use a lambda expression and static scope  (defun REPTOP (m r L)  (mapcar  (lambda (value)  (COND ( (EQL value m) r)  ( T value )  )  )  L ;;; second parameter to mapcar  )  )  Lambda expression embedded inside reptop-> now knows about m and r |
| **Python Iterators**  The for statement in Python can iterate through a *containingObject* returning an item from that *containingObject* on each iteration.  for *controlVar* in *containingObject*:  *indentedBodyStatements*  To create a class which can be accessed by a for statement as above, the class must support the iterator protocol. It must provide this method:  **\_\_iter\_\_(self):**  returns an **iterator** object which can be the object itself or a separate object which acts as an iteration cursor, stepping through the items  The class that is the iterator must provide this method:  **\_\_next\_\_(self):**  returns the next item in the iteration or raises the **StopIteration** exception to end the iteration  Python provides an iterator protocol which describes methods that must be supported by a class. | # Python Example P-1  fruitM = ["apple", "orange", "banana"]  for fruit in fruitM:  print (fruit)  Output:  apple  orange  banana  # Python Example P-2  contactM = {"Board, Bill": "830-222-2222"  , "Board, Peg": "830-222-3333"  , "Barr, Ted E": "210-555-1111"}  contactM["Board, Emory"] = contactM["Board, Peg"]; #copy Peg's phone to Emory  # access the key and value, calling them name and phone  for name, phone in contactM.items():  print (name, phone)  Output:  Board, Emory 830-222-3333  Board, Bill 830-222-2222  Barr, Ted E 210-555-1111  Board, Peg 830-222-3333 |
| The Counter class in the example on the right provides the iteration cursor which gives numbers from a lowValue to a highValue, inclusive. | # Python Example P-3  # Counter.py  # Provides a counter from a low value to a high value  class Counter:  def \_\_init\_\_(self, lowValue, highValue):  self.current = lowValue  self.high = highValue  def \_\_iter\_\_(self):  return self # Counter, itself, is the iteration cursor  def \_\_next\_\_(self):  if self.current > self.high:  raise StopIteration #causes a for to complete  else:  self.current +=1  return self.current - 1    for cnt in Counter(1, 5):  print (cnt)  1  2  3  4  5 |
|  | # Python Example P-4  # Reverse.py  # Reverse provides an iterator that delivers the contents of a list in reverse  class Reverse:  def \_\_init\_\_(self, list):  self.list = list  self.current = len(self.list)  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.current == 0:  raise StopIteration  self.current = self.current - 1  return self.list[self.current]  for item in Reverse(["s", "t", "r", "e", "s", "s", "e", "d"]):  print (item, end=" ")  print(" ")  **d e s s e r t s** |
| **Python Generator**  A generator is a function that produces a sequence of results. Instead of return statements, they yield the next value in the sequence using the **yield** statement.  A generator function uses the **yield *value*** statement to provide the next value, but doesn't exit the function. (This is like a coroutine.) It retains all local context and resumes execution on the next call. | # Python Example P-5  # genReverse is a generator that yields the contents of a list  # in reverse order. Notice that it stays in the loop when it  # yields a value.  def genReverse(list):  current = len(list)  while current > 0:  current = current - 1  yield list[current]    for item in genReverse(["s", "t", "r", "e", "s", "s", "e", "d"]):  print (item, end=" ")  print(" ")  **d e s s e r t s** |
| **Generator which doesn't first have the list of values**  what would happen if the invoking for loop didn't check for 20?  Infinite loop | # Python Example P-6  # genPrimes is a generator that yields prime numbers  >>> def genPrimes():  n = 2  primeM = [] # start with an empty list of previously known primes  while True:  # see if one of the previous primes can be divided into  # our next attempt, n, without a remainder  for prime in primeM:  if n % prime == 0:  break # n isn't a prime  else: # if we fell out of the for loop normally, n is a prime  primeM.append(n)  yield n  # get the next attempt  n += 1  # print the first 20 prime numbers  count = 0  for primeNum in genPrimes():  count += 1  if count > 20:  break  print (primeNum, end = " ")  **Output:**  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 |
| **For safety, generators should have a termination condition.** | >>> def genPrimes(maxValue=1000):  n = 2  primeM = [] # start with an empty list of previously known primes  while n <= maxValue:  # see if one of the previous primes can be divided into  # our next attempt, n, without a remainder  for prime in primeM:  if n % prime == 0:  break # n isn't a prime  else: # if we fell out of the for loop normally, n is a prime  primeM.append(n)  yield n  # get the next attempt  n += 1  # print the prime numbers until we reach the maxValue  for primeNum in genPrimes():  print (primeNum, end= " ")  **Output:**  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 |
| **Java Iterators**  If a class properly supports methods as an iterator, it can be referenced in a for statement.  We already saw Java for statements which iterate through items in a containing object.  for (*datatype itemVar* : *containingObject*)  *bodyStatements*;  The Iterator interface describes what is needed to support the necessary behavior in the *containingObject.*  public interface Iterator<E>  {  E next(); // returns next instance  Boolean hasNext();  void remove();  }  In addition to those three methods, the *containingObject* must provide this method for creating an Iterator:  public Iterator<E> iterator() | // Java Example Iterator for returning the reverse of an Array  import java.util.Iterator;  import java.lang.reflect.\*; // contains the Array class  public class ReverseIterator<T> implements Iterator<T>, Iterable<T>  {  private T array[];  private int current;    public ReverseIterator(T array[])  {  this.array = array;  this.current = Array.getLength(array);  }  public Iterator<T> iterator()  {  return this;  }  public boolean hasNext()  {  return current > 0;  }  public T next()  {  current--;  return array[current];  }  public void remove()  {  throw new UnsupportedOperationException();  }  } |
| The code on the right uses that ReverseIterator in Java | import java.util.\*;  public class RevMainClass  {  static String arrayM[] = {"s", "t", "r", "e", "s", "s", "e", "d"};    public static void main(String [] args)  {  for ( String str : new ReverseIterator<String>(arrayM))  System.out.println(str);  }  } |
|  |  |
|  |  |