```
def factors(n):              # traditional function that computes factors


   results = []             # store factors in a new list


   for k in range(1,n+1):    #
      if n%k == 0:         # divides evenly; thus k is a factor
         results.append(k) # append k to the list of factors


   return results          # return the entire list




# An iterator is an object that manages an iteration through a series of values.

# If variable, o, identifies an interator object, then each call to the built-in function , next(o),

# produces a subsequent element from the underlying series, with a StopIteration exception

# raised to indicate that there are no further elements.


# An iterable is an object , obj, that produces an iterator via the syntax iter(obj).

# An instance of a list is an iterable, but not itself an iterator.

# With data = [1,2,4,8], it is not legal to call next(data).

# iterator object can be produced with the syntax  o = iter(data) and then each subsequent call to
next(o)

# will return an element of the list.


# The for loop syntax in Python simply automates the above memntioned process, creating an iterator
for the

# given iterable, and then repeatedly calling for the next element until catching the StopIteration

# exception.
```

```
#Concept of Generators:

# Python also supports functions and classes that produce an implicit iterable series of values,
# that is, without constructing a data structure to store all of its values at once.
# For example, the call range(1000000) does not return a list of numbers , it returns a range
# object that is iterable.
# This object generates the million values , one at a time, and only as needed.
# The above is termed  LAZY EVALUATION.
# In the case of range, it allows a loop of the form, for j in range(1000000):, to execute without
# setting aside memory for storing one million values.
```

```
# For example, the call range(1000000) does not return a list of numbers , it returns a range
# object that is iterable.
# This object generates the million values , one at a time, and only as needed.
# The above is termed  LAZY EVALUATION.
# In the case of range, it allows a loop of the form, for j in range(1000000):, to execute without
# setting aside memory for storing one million values.
```

```
def lazyFactors(n):             # generator that computes factors

   for k in range(1,n+1):   #

      if n%k == 0:        # divides evenly, thus k is a factor
         yield k         # yield this factor as next result
```

```
def fibonacci_generator():

    a=0
    b=1
    while True:        #keep computing forever!  #potential of infinite computation
        yield a        #report value, a , during the current pass
        future = a+b
        a=b            # next value that will be reported
        b=future       # and subsequently this one

for i in fibonacci_generator():

    print(i)
    if (i>1000):

        break
```

```
def factors(n):

    k=1
    while k*k < n:

        if n%k == 0:

            yield k
            yield n//k
        k = k+1

        if k*k == n:

            yield k
```