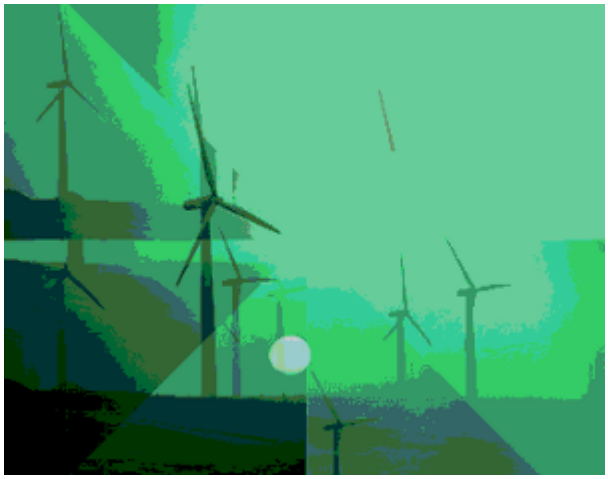
GENERATORS

INTRODUCTION



An iterator can be seen as a pointer to a container, e.g. a list structure that can iterate over all the elements of this container. The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object. In some object oriented programming languages, like Perl, Java and Python, iterators are implicitly available and can be used in foreach loops, corresponding to for loops in Python.

Generators are a special kind of function, which enable us to implement or generate iterators.

Iterators are a fundamental concept of Python.
Mostly, iterators are implicitly used, like in the for-loop of Python. We demonstrate this in the following example. We are iterating over a list, but you shouldn't be mistaken: A list is not an iterator, but it can be used like an iterator:

```
>>> cities = ["Paris", "Berlin", "Hamburg", "Frankfurt", "London",
"Vienna", "Amsterdam", "Den Haag"]
>>> for location in cities:
...     print("location: " + location)
...
location: Paris
location: Berlin
location: Hamburg
location: Frankfurt
location: London
location: Vienna
location: Amsterdam
location: Den Haag
>>>
```

What is really happening, when you use an iterable like a string, a list, or a tuple, inside of a for loop is the following: The function "iter" is called on the iterable. The return value of iter is an iterable. We can iterate over this iterable with the next function until the iterable is exhausted and returns a StopIteration exception:

```
>>> expertises = ["Novice", "Beginner", "Intermediate",
"Proficient", "Experienced", "Advanced"]
>>> expertises_iterator = iter(expertises)
>>> next(expertises_iterator)
'Novice'
>>> next(expertises_iterator)
'Beginner'
>>> next(expertises_iterator)
'Intermediate'
>>> next(expertises_iterator)
'Proficient'
>>> next(expertises_iterator)
'Experienced'
>>> next(expertises_iterator)
'Advanced'
>>> next(expertises_iterator)
Traceback (most recent call last):
  File "", line 1, in
StopIteration
```

Internally, the for loop also calls the next function and terminates, when it gets StopIteration.

We can simulate this iteration behavior of the for loop in a while loop: You might have noticed that there is something missing in our program: We have to catch the "Stop Iteration" exception:

```
other_cities = ["Strasbourg", "Freiburg", "Stuttgart",
                "Vienna / Wien", "Hannover", "Berlin",
                "Zurich"]

city_iterator = iter(other_cities)
while city_iterator:
    try:
        city = next(city_iterator)
        print(city)
    except StopIteration:
        break
```

We get the following output from this program:

```
Strasbourg
Freiburg
Stuttgart
Vienna / Wien
Hannover
Berlin
Zurich
```

The sequential base types as well as the majority of the classes of the standard library of Python support iteration. The dictionary data type dict supports iterators as well. In this case the iteration runs over the keys of the dictionary:

```
>>> capitals = { "France":"Paris", "Netherlands":"Amsterdam",
"Germany":"Berlin", "Switzerland":"Bern", "Austria":"Vienna"}
>>> for country in capitals:
...     print("The capital city of " + country + " is " +
capitals[country])
...
The capital city of Switzerland is Bern
The capital city of Netherlands is Amsterdam
The capital city of Germany is Berlin
The capital city of France is Paris
>>>
```

Off-topic: Some readers may be confused to learn from our example that the capital of the Netherlands is not Den Haag (The Hague) but Amsterdam. Amsterdam is the capital of the Netherlands according to the constitution, even though the Dutch parliament and the Dutch government are situated in The Hague, as well as the Supreme Court and the Council of State.

GENERATORS

On the surface generators in Python look like functions, but there is both a syntactic and a semantic difference. One distinguishing characteristic is the yield statements. The yield statement turns a functions into a generator. A generator is a function which returns a generator object. This generator object can be seen like a function which produces a sequence of results instead of a single object. This sequence of values is produced by iterating over it, e.g. with a for loop. The values, on which can be iterated, are created by using the yield statement. The value created by the yield statement is the value following the yield keyword. The execution of the code stops when a yield statement has been reached. The value behind the yield will be returned. The execution of the generator is interrupted now. As soon as "next" is called again on the generator object, the generator function will resume execution right after the yield statement in the code, where the last call exited. The execution will continue in the state in which the generator was left after the last yield. This means that all the local variables still exists, because they are automatically saved between calls. This is a fundamental difference to functions: functions always start their execution at the beginning of the function body, regardless where they had left in previous calls. They don't have any static or persistent values. There may be more than one yield statement in the code of a generator or the yield statement might be inside the body of a loop. If there is a return statement in the code of a generator, the execution will stop with a StopIteration exception error if this code is executed by the Python interpreter. The word "generator" is sometimes ambiguously used to mean both the generator function itself and the objects which are generated by a generator.

Everything which can be done with a generator can also be implemented with a class based iterator as well. But the crucial advantage of generators consists in automatically creating the methods __iter__()

and next().
Generators provide a very neat way of producing data which is huge or infinite.

The following is a simple example of a generator, which is capable of producing various city names:

```
def city_generator():
    yield("London")
    yield("Hamburg")
    yield("Konstanz")
    yield("Amsterdam")
    yield("Berlin")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

It's possible to create a generator object with this generator, which generates all the city names, one after the other:

```
>>> from city_generator import city_generator
>>> city = city_generator()
>>> print(next(city))
London
>>> print(next(city))
Hamburg
>>> print(next(city))
Konstanz
>>> print(next(city))
Amsterdam
>>> print(next(city))
Berlin
>>> print(next(city))
Zurich
>>> print(next(city))
Schaffhausen
>>> print(next(city))
Stuttgart
>>> print(next(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

As we can see, we have generated an iterator x in the interactive shell. Every call of the method next() returns another city. After the last city, i.e. Stuttgart, has been created, another call of next(x) raises an error, saying that the iteration has stopped, i.e. "StopIteration".
Can we send a reset to an iterator is a frequently asked question, so that it can start the iteration all over again. There is no reset, but it's possible to create another generator. This can be done e.g. by having the statement "x = city_generator()" again.
Though at first sight the yield statement looks like the return statement of a function, we can see in this

example that there is a big difference. If we had a return statement instead of a yield in the previous example, it would be a function. But this function would always return "London" and never any of the other cities, i.e. "Hamburg", "Konstanz", "Amsterdam", "Berlin", "Zurich", "Schaffhausen", and "Stuttgart"

METHOD OF OPERATION

As we have elaborated in the introduction of this chapter, the generators offer a comfortable method to generate iterators, and that's why they are called generators.

Method of working:

- A generator is called like a function. Its return value is an iterator, i.e. a generator object. The code of the generator will not be executed at this stage.
- The iterator can be used by calling the next method. The first time the execution starts like a function, i.e. the first line of code within the body of the iterator. The code is executed until a yield statement is reached.
- yield returns the value of the expression, which is following the keyword yield. This is like a function, but Python keeps track of the position of this yield and the state of the local variables is stored for the next call. At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call.
- The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value.

We will illustrate this behaviour in the following example, in which we define a generator which generates an iterator for all the Fibonacci numbers.
The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of filius Bonacci, "son of Bonaccio"). In his textbook Liber Abaci, which appeared in the year 1202) he had an exercise about the rabbits and their breeding: It starts with a newly-born pair of rabbits, i.e. a male and a female animal. It takes one month until they can mate. At the end of the second month the female gives birth to a new pair of rabbits. Now let's suppose that every female rabbit will bring forth another pair of rabbits every month after the end of the first month. We have to mention that Fibonacci's rabbits never die. They question is how large the population will be after a certain period of time.

This produces a sequence of numbers: 0,1,1,2,3,5,8,13

This sequence can be defined in mathematical terms like this:

$F_n = F_{n-1} + F_{n-2}$
with the seed values:
$F_0 = 0$ and $F_1 = 1$

```
def fibonacci(n):
    """ A generator for creating the Fibonacci numbers """
    a, b, counter = 0, 1, 0
    while True:
```

```
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5)
for x in f:
    print(x, " ", end="") #
print()
```

The generator above can be used to create the first n Fibonacci numbers separated by blanks, or better (n+1) numbers because the 0th number is also included.
In the next example we present a version which is capable of returning an endless iterator. We have to take care when we use this iterator that a termination criterion is used:

```
def fibonacci():
    """Generates an infinite sequence of Fibonacci numbers on
demand"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

f = fibonacci()

counter = 0
for x in f:
    print(x, " ", end="")
    counter += 1
    if (counter > 10):
        break
print()
```

USING A 'RETURN' IN A GENERATOR

Since Python 3.3, generators can also use return statements, but a generator still needs at least one yield statement to be a generator! A return statement inside of a generator is equivalent to `raise StopIteration()`

Let's have a look at a generator in which we raise StopIteration:

```
>>> def gen():
...     yield 1
...     raise StopIteration(42)
...
```

```
>>>
>>> g = gen()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "", line 1, in
  File "", line 3, in gen
StopIteration: 42
>>>
```

We demonstrate now that return is equivalent, or "nearly", if we disregard one line of the traceback:

```
>>> def gen():
...     yield 1
...     return 42
...
>>>
>>> g = gen()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "", line 1, in
StopIteration: 42
>>>
```

SEND METHOD /COROUTINES

Generators can not only send objects but also receive objects. Sending a message, i.e. an object, into the generator can be achieved by applying the send method to the generator object. Be aware of the fact that send both sends a value to the generator and returns the value yielded by the generator. We will demonstrate this behavior in the following simple example of a coroutine:

```
>>> def simple_coroutine():
...     print("coroutine has been started!")
...     x = yield
...     print("coroutine received: ", x)
...
>>> cr = simple_coroutine()
>>> cr

>>> next(cr)
coroutine has been started!
>>> cr.send("Hi")
```

```
coroutine received:  Hi
Traceback (most recent call last):
  File "", line 1, in
StopIteration
>>>
```

We had to call next on the generator first, because the generator needed to be started. Using send to a generator which hasn't been started leads to an exception:

```
>>> cr = simple_coroutine()
>>> cr.send("Hi")
Traceback (most recent call last):
  File "", line 1, in
TypeError: can't send non-None value to a just-started generator
>>>
```

To use the send method the generator has to wait at a yield statement, so that the data sent can be processed or assigned to the variable on the left side. What we haven't said so far: A next call also sends and receives. It always sends a None object. The values sent by "next" and "send" are assigned to a variable within the generator: this variable is called "message" in the following example. We called the generator infinit_looper, because it takes a sequential data objects and creates an iterator, which is capable of looping forever over the object, i.e. it starts again with the first element after having delivered the last object. By sending an index to the iterator, we can continue at an arbitrary position.

```
def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        message = yield objects[count]
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We demonstrate how to use this generator in the following interactive session, assuming that the generator is saved in a file called generator_decorator.py:

```
>>> x = infinite_looper("A string with some words")
>>> next(x)
'A'
>>> x.send(9)
'w'
>>> x.send(10)
'i'
>>>
```

THE THROW METHOD

The throw() method raises an exception at the point where the generator was paused, and returns the next value yielded by the generator. It raises StopIteration if the generator exits without yielding another value. The generator has to catch the passed-in exception, otherwise the exception will be propagated to the caller. The infinite_looper from our previous example keeps yielding the elements of the sequential data, but we don't have any information about the index or the state of the variable "count". We can get this information by throwing an exception with the "throw" method. We catch this exception inside of the generator and print the value of "count":

```
def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        try:
            message = yield objects[count]
        except Exception:
            print("index: " + str(count))
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We can use it like this:

```
>>> from generator_throw import infinite_looper
>>> looper = infinite_looper("Python")
>>> next(looper)
'P'
>>> next(looper)
'y'
>>> looper.throw(Exception)
index: 1
't'
>>> next(looper)
'h'
```

We can improve the previous example by defining our own exception class StateOfGenerator:

```
class StateOfGenerator(Exception):
     def __init__(self, message=None):
         self.message = message

def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        try:
            message = yield objects[count]
```

```
        except StateOfGenerator:
            print("index: " + str(count))
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We can use the previous generator like this:

```
>>> from generator_throw import infinite_looper, StateOfGenerator
>>> looper = infinite_looper("Python")
>>> next(looper)
'P'
>>> next(looper)
'y'
>>> looper.throw(StateOfGenerator)
index: 1
't'
>>> next(looper)
'h'
>>>
```

DECORATING GENERATORS

There is one problem with our approach, we cannot start the iterator by sending directly an index to it. Before we can do this, we need to use the next function to start the iterator and advance it to the yield statement. We will write a decorator now, which can be used to make a decorator ready, by advancing it automatically at creation time to the yield statement. This way, it will be possible to use the send method directly after initialisation of a generator object.

```
from functools import wraps

def get_ready(gen):
    """
    Decorator: gets a generator gen ready
    by advancing to first yield statement
    """
    @wraps(gen)
    def generator(*args,**kwargs):
        g = gen(*args,**kwargs)
        next(g)
        return g
    return generator

@get_ready
def infinite_looper(objects):
    count = -1
```

```
    message = yield None
    while True:
        count += 1
        if message != None:
            count = 0 if message < 0 else message
        if count >= len(objects):
            count = 0
        message = yield objects[count]


x = infinite_looper("abcdef")
print(next(x))
print(x.send(4))
print(next(x))
print(next(x))
print(x.send(5))
print(next(x))
```

This program returns the following results:

```
a
e
f
a
f
a
```

You might have noticed that we have changed the generator infinite_looper a little bit as well.

YIELD FROM

"yield from" is available since Python 3.3!
The `yield from <expr>` statement can be used inside the body of a generator. `<expr>` has to be an expression evaluating to an iterable, from which an iterator will be extracted.
The iterator is run to exhaustion, i.e. until it encounters a `StopIteration` exception. This iterator yields and receives values to or from the caller of the generator, i.e. the one which contains the `yield from` statement.

We can learn from the following example by looking at the two generators 'gen1' and 'gen2' that yield from is substituting the for loops of 'gen1':

```
def gen1():
    for char in "Python":
        yield char
    for i in range(5):
        yield i

def gen2():
```

```
    yield from "Python"
    yield from range(5)

g1 = gen1()
g2 = gen2()
print("g1: ", end=", ")
for x in g1:
    print(x, end=", ")
print("\ng2: ", end=", ")
for x in g2:
    print(x, end=", ")
print()
```

We can see from the output that both generators are the same:

```
g1: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
g2: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
```

The benefit of a `yield from` statement can be seen as a way to split a generator into multiple generators. That's what we have done in our previous example and we will demonstrate this more explicitely in the following example:

```
def cities():
    for city in ["Berlin", "Hamburg", "Munich", "Freiburg"]:
        yield city

def squares():
    for number in range(10):
        yield number ** 2

def generator_all_in_one():
    for city in cities():
        yield city
    for number in squares():
        yield number

def generator_splitted():
    yield from cities()
    yield from squares()

lst1 = [el for el in generator_all_in_one()]
lst2 = [el for el in generator_splitted()]
print(lst1 == lst2)
```

The previous code returns `True` because the generators `generator_all_in_one` and `generator_splitted` yield the same elements. This means that if the `<expr>` from the `yield from` is another generator, the effect is the same as if the body of the sub-generator were inlined at the point of the `yield from` statement. Furthermore, the subgenerator is allowed to execute a return statement

with a value, and that value becomes the value of the yield from expression. We demonstrate this with the following little script:

```
def subgenerator():
    yield 1
    return 42

def delegating_generator():
    x = yield from subgenerator()
    print(x)

for x in delegating_generator():
    print(x)
```

The above code returns the following code:

```
1
42
```

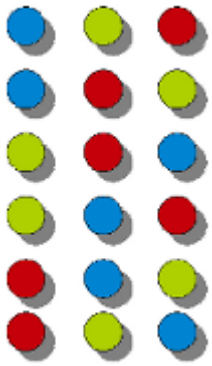
The full semantics of the `yield from` expression is described in six points in *"PEP 380 -- Syntax for Delegating to a Subgenerator"* in terms of the generator protocol:

- Any values that the iterator yields are passed directly to the caller.
- Any values sent to the delegating generator using `send()` are passed directly to the iterator. If the sent value is None, the iterator's `__next__()` method is called. If the sent value is not None, the iterator's `send()` method is called. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
- Exceptions other than `GeneratorExit` thrown into the delegating generator are passed to the `throw()` method of the iterator. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
- If a `GeneratorExit` exception is thrown into the delegating generator, or the `close()` method of the delegating generator is called, then the `close()` method of the iterator is called if it has one. If this call results in an exception, it is propagated to the delegating generator. Otherwise, `GeneratorExit` is raised in the delegating generator.
- The value of the yield from expression is the first argument to the StopIteration exception raised by the iterator when it terminates.
- `return expr` in a generator causes `StopIteration(expr)` to be raised upon exit from the generator.

RECURSIVE GENERATORS

Like functions generators can be recursively programmed. The following example is a generator to create all the permutations of a given list of items.

For those who don't know what permutations are, we have a short introduction:



Formal Definition:
A permutation is a rearrangement of the elements of an ordered list. In other words: Every arrangement of n elements is called a permutation.

In the following lines we show you all the permutations of the letter a, b and c:

a b c
a c b
b a c
b c a
c a b
c b a

The number of permutations on a set of n elements is given by n!
n! = n*(n-1)*(n-2) ... 2 * 1
n! is called the factorial of n.

The permutation generator can be called with an arbitrary list of objects. The iterator returned by this generator generates all the possible permutations:

```
def permutations(items):
    n = len(items)
    if n==0: yield []
    else:
        for i in range(len(items)):
            for cc in permutations(items[:i]+items[i+1:]):
                yield [items[i]]+cc

for p in permutations(['r','e','d']): print(''.join(p))
for p in permutations(list("game")): print(''.join(p) + ", ",
end="")
```

The previous example can be hard to understand for newbies. As often, Python offers a convenient solution. We need the module itertools for this purpose. Itertools is a very handy tool to create and operate on iterators.

Creating permutations with itertools:

```
>>> import itertools
>>> perms = itertools.permutations(['r','e','d'])
>>> perms
<itertools.permutations object at 0x7fb0da3e4a70>
>>> list(perms)
[('r', 'e', 'd'), ('r', 'd', 'e'), ('e', 'r', 'd'), ('e', 'd',
'r'), ('d', 'r', 'e'), ('d', 'e', 'r')]
>>>
```

The term "permutations" can sometimes be used in a weaker meaning. Permutations can denote in this weaker meaning a sequence of elements, where each element occurs just once, but without the requirement to contain all the elements of a given set. So in this sense (1,3,5,2) is a permutation of the set of digits {1,2,3,4,5,6}. We can build for example all the sequences of a fixed length k of elements taken from a given set of size n with $k \leq n$.

These are are all the 3-permutations of the set {"a","b","c","d"}:

```
['a', 'b', 'c']
['a', 'b', 'd']
['a', 'c', 'b']
['a', 'c', 'd']
['a', 'd', 'b']
['a', 'd', 'c']
['b', 'a', 'c']
['b', 'a', 'd']
['b', 'c', 'a']
['b', 'c', 'd']
['b', 'd', 'a']
['b', 'd', 'c']
['c', 'a', 'b']
['c', 'a', 'd']
['c', 'b', 'a']
['c', 'b', 'd']
['c', 'd', 'a']
['c', 'd', 'b']
['d', 'a', 'b']
['d', 'a', 'c']
['d', 'b', 'a']
['d', 'b', 'c']
['d', 'c', 'a']
['d', 'c', 'b']
```

These atypical permutations are also known as **sequences without repetition**. By using this term we can avoid confusion with the term "permutation". The number of such k-permutations of n is denoted by $P_{n,k}$ and its value is calculated by the product:
$n \cdot (n - 1) \cdot \ldots (n - k + 1)$
By using the factorial notation, the above expression can be written as:

$$P_{n,k} = n! / (n - k)!$$

A generator for the creation of k-permuations of n objects looks very similar to our previous permutations generator:

```
def k_permutations(items, n):
    if n==0:
        yield []
```

```
        else:
            for item in items:
                for kp in k_permutations(items, n-1):
                    if item not in kp:
                        yield [item] + kp

for kp in k_permutations("abcd", 3):
    print(kp)
```

The above program returns the following k-permutations:

```
['a', 'b', 'c']
['a', 'b', 'd']
['a', 'c', 'b']
['a', 'c', 'd']
['a', 'd', 'b']
['a', 'd', 'c']
['b', 'a', 'c']
['b', 'a', 'd']
['b', 'c', 'a']
['b', 'c', 'd']
['b', 'd', 'a']
['b', 'd', 'c']
['c', 'a', 'b']
['c', 'a', 'd']
['c', 'b', 'a']
['c', 'b', 'd']
['c', 'd', 'a']
['c', 'd', 'b']
['d', 'a', 'b']
['d', 'a', 'c']
['d', 'b', 'a']
['d', 'b', 'c']
['d', 'c', 'a']
['d', 'c', 'b']
{'c', 'd', 'e'}
{'a', 'd', 'e'}
{'f', 'd', 'e'}
{'b', 'd', 'e'}
{'f', 'c', 'd'}
{'c', 'a', 'd'}
{'b', 'c', 'd'}
{'f', 'a', 'd'}
{'b', 'a', 'd'}
{'b', 'd', 'f'}
```

```
{'c', 'a', 'e'}
{'f', 'a', 'e'}
{'b', 'a', 'e'}
{'f', 'c', 'a'}
{'b', 'c', 'a'}
{'b', 'a', 'f'}
{'f', 'c', 'e'}
{'b', 'c', 'e'}
{'b', 'c', 'f'}
{'b', 'f', 'e'}
```

A GENERATOR OF GENERATORS

The second generator of our Fibonacci sequence example generates an iterator, which can theoretically produce all the Fibonacci numbers, i.e. an infinite number. But you shouldn't try to produce all these numbers, as we would do in the following example:

```
list(fibonacci())
```

This will show you very fast the limits of your computer.
In most practical applications, we only need the first n elements of an "endless" iterator. We can use another generator, in our example firstn, to create the first n elements of a generator g:

```
def firstn(g, n):
        for i in range(n):
                yield next(g)
```

The following script returns the first 10 elements of the Fibonacci sequence:

```
#!/usr/bin/env python3
def fibonacci():
    """Ein Fibonacci-Zahlen-Generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def firstn(g, n):
        for i in range(n):
                yield next(g)

print(list(firstn(fibonacci(), 10)))
```

The output looks like this:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

EXERCISES

1. Write a generator which computes the running average.
2. Write a generator "trange", which generates a sequence of time tuples from start to stop incremented by step. A time tuple is a 3-tuple of integers: (hours, minutes, seconds)
   Example:

   ```
   for time in trange((10, 10, 10), (13, 50, 15), (0, 15, 12) ):
           print(time)
   ```

   will return

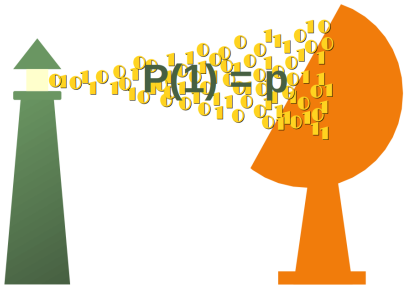
   ```
   (10, 10, 10)
   (10, 25, 22)
   (10, 40, 34)
   (10, 55, 46)
   (11, 10, 58)
   (11, 26, 10)
   (11, 41, 22)
   (11, 56, 34)
   (12, 11, 46)
   (12, 26, 58)
   (12, 42, 10)
   (12, 57, 22)
   (13, 12, 34)
   (13, 27, 46)
   (13, 42, 58)
   ```

3. Write a version "rtrange" of the previous generator, which can receive message to reset the start value.

4. Write a program, using the newly written generator "trange", to create a file "times_and_temperatures.txt". The lines of this file contain a time in the format hh::mm::ss and random temperatures between 10.0 and 25.0 degrees. The times should be ascending in steps of 90 seconds starting with 6:00:00.
   For example:

```
06:00:00 20.1
06:01:30 16.1
06:03:00 16.9
06:04:30 13.4
06:06:00 23.7
06:07:30 23.6
06:09:00 17.5
06:10:30 11.0
```

5. Write a generator with the name "random_ones_and_zeroes", which returns a bitstream, i.e. a zero or a one in every iteration. The probability p for returning a 1 is defined in a variable p. The generator will initialize this value to 0.5. This means that zeroes and ones will be returned with the same probability.



SOLUTIONS TO OUR EXERCISES

1.
```
def running_average():
    total = 0.0
    counter = 0
    average = None
    while True:
        term = yield average
        total += term
        counter += 1
        average = total / counter


ra = running_average()  # initialize the coroutine
next(ra)                # we have to start the coroutine
for value in [7, 13, 17, 231, 12, 8, 3]:
    out_str = "sent: {val:3d}, new average: {avg:6.2f}"
    print(out_str.format(val=value, avg=ra.send(value)))
```

2.
```
def trange(start, stop, step):
    """
    trange(stop) -> time as a 3-tuple (hours, minutes,
```

```
    seconds)
        trange(start, stop[, step]) -> time tuple

        start: time tuple (hours, minutes, seconds)
        stop: time tuple
        step: time tuple

        returns a sequence of time tuples from start to stop
    incremented by step
        """

        current = list(start)
        while current < list(stop):
            yield tuple(current)
            seconds = step[2] + current[2]
            min_borrow = 0
            hours_borrow = 0
            if seconds < 60:
                current[2] = seconds
            else:
                current[2] = seconds - 60
                min_borrow = 1
            minutes = step[1] + current[1] + min_borrow
            if minutes < 60:
                current[1] = minutes
            else:
                current[1] = minutes - 60
                hours_borrow = 1
            hours = step[0] + current[0] + hours_borrow
            if hours < 24:
                current[0] = hours
            else:
                current[0] = hours -24

    if __name__ == "__main__":
        for time in trange((10, 10, 10), (13, 50, 15), (0, 15, 12)
    ):
            print(time)
```

3.
```
    def rtrange(start, stop, step):
        """
        trange(stop) -> time as a 3-tuple (hours, minutes,
    seconds)
        trange(start, stop[, step]) -> time tuple
```

```
        start: time tuple (hours, minutes, seconds)
        stop: time tuple
        step: time tuple

        returns a sequence of time tuples from start to stop
    incremented by step

        The generator can be rest by sending a new "start" value.
        """

        current = list(start)
        while current < list(stop):
            new_start = yield tuple(current)
            if new_start != None:
                current = list(new_start)
                continue
            seconds = step[2] + current[2]
            min_borrow = 0
            hours_borrow = 0
            if seconds < 60:
                current[2] = seconds
            else:
                current[2] = seconds - 60
                min_borrow = 1
            minutes = step[1] + current[1] + min_borrow
            if minutes < 60:
                current[1] = minutes
            else:
                current[1] = minutes - 60
                hours_borrow = 1
            hours = step[0] + current[0] + hours_borrow
            if hours < 24:
                current[0] = hours
            else:
                current[0] = hours -24

    if __name__ == "__main__":
        ts = rtrange((10, 10, 10), (13, 50, 15), (0, 15, 12) )
        for _ in range(3):
            print(next(ts))

        print(ts.send((8, 5, 50)))
        for _ in range(3):
            print(next(ts))
```

Calling this program will return the following output:

```
(10, 10, 10)
(10, 25, 22)
(10, 40, 34)
(8, 5, 50)
(8, 21, 2)
(8, 36, 14)
(8, 51, 26)
```

4.
```
from timerange import trange
import random

fh = open("times_and_temperatures.txt", "w")

for time in trange((6, 0, 0), (23, 0, 0), (0, 1, 30) ):
    random_number = random.randint(100, 250) / 10
    lst = time + (random_number,)
    output = "{:02d}:{:02d}:{:02d} {:4.1f}\n".format(*lst)
    fh.write(output)
```

5. You can find further details and the mathematical background about this exercise in our chapter on Weighted Probabilities.

```
import random

def random_ones_and_zeros():
    p = 0.5
    while True:
        x = random.random()
        message = yield 1 if x < p else 0
        if message != None:
            p = message

x = random_ones_and_zeros()
next(x)  # we are not interested in the return value
for p in [0.2, 0.8]:
    print("\nWe change the probability to : " + str(p))
    x.send(p)
    for i in range(20):
        print(next(x), end=" ")
print()
```

We get the following output:

```
We change the probability to : 0.2
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
```

```
We change the probability to : 0.8
1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```