

Extended Introduction to Computer Science

CS1001.py

Module G Generators for Streams

Instructors: Elhanan Borenstein, Michal Kleinbort

Teaching Assistants: Noam Parzanchevsky,

Omri Porat, Moshe Shechner

School of Computer Science

Tel-Aviv University

<http://tau-cs1001-py.wikidot.com>

* Slides based on a course designed by Prof. Benny Chor

Data Structures

1. Linked Lists ✓
2. Binary Search Trees ✓
3. Hash tables ✓
4. Generators ←

Generators for Infinite Streams

- So far we handled **finite** (and **not too large**) data collections.
- We now change the setting a bit, and talk about **infinite collections** of data, aka **streams**. For example:
 - A **router** that handles incoming **internet packages** over time. Input just keeps coming, and the router needs to direct it **"on the fly"**.
 - The **"Secretary Problem"**: a manager interviews applicants in random order, and she makes a decision being unaware of the quality of **yet unseen applicants**
 - Computing **"moving average"** (of e.g. daily temperature over the month)
- Algorithms whose input is an endless data stream are called **online algorithm** (as opposed to **offline** algorithms whose **entire input** is given at the function call, which was the standard scenario assumed so far)
- Obviously, we cannot store the entire stream simultaneously
- In Python, such streams can be represented using **generators**, which are created by **generator functions**.

Example1: Natural Number Generator

```
def naturals():  
    n = 0  
    while True:  
        yield n  
        n+=1
```

- Any function that contains a **yield** statement is termed a **generator function**.
- When a generator function is called, no code in the body of the function is executed. Instead, a **generator object** is returned.

```
>>> nat = naturals()  
>>> type(nat)  
<class 'generator'>
```

Example1: Natural Number Generator (cont.)

- `nat` is a generator. To extract the “next” value in the stream it represents, which is specified by the `yield` statement, we invoke Python’s built-in function `next`:

```
>>> nat = naturals()
>>> next(nat)
0
>>> next(nat)
1
>>> [next(nat) for i in range(10)]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
def naturals():
    n = 0
    while True:
        yield n
        n+=1
```

- We see that `nat` has a `state`, which is retained, unchanged, between successive calls to `next`.
- We can have additional instances of the generator:

```
>>> nat2 = naturals()
>>> next(nat2)
0
>>> next(nat)
12
```

Example1: Natural Number Generator (cont.)

```
def naturals():  
    n = 0  
    while True:  
        yield n  
        n+=1
```

```
nat = naturals()  
for i in range(10):  
    print(next(nat))
```

0
1
2
3
4
5
6
7
8
9

```
for i in range(10):  
    print(next(naturals()))
```

0
0
0
0
0
0
0
0
0
0

Execution Specification

- So far, our functions contained no **state**, or **memory**: successive calls to the function with the same arguments produced the same results (assuming the function is deterministic, and does not refer to a global variable, which may have changed).

- In contrast, when a **yield** statement of the form

`yield expression`

is encountered, the **state** of the function is "**frozen**", and the value of `expression` is returned to the caller of `next`.

- By "frozen" we mean that all **local state is retained**, including the current bindings of local variables, the instruction pointer (pointing at the next instruction to execute), and the internal evaluation stack
- Enough information is saved so that the next time `next` is invoked, the function can proceed exactly as if the **yield** statement were just another external call.

(see <http://www.python.org/dev/peps/pep-0255/>)

Lazy Evaluation

- In programming language theory, **lazy evaluation** or **call-by-need** is an evaluation strategy, which **delays** the evaluation of an expression until its value is actually required.
- Python's **generators** employ lazy evaluation. The next item is evaluated only when it is required, by means of executing `next()`.
- The "opposite" of lazy actions is **eager evaluation**, sometimes known as **strict evaluation**. Eager evaluation is the evaluation behavior used in most cases in most programming languages.

Example1.5: A Fibonacci Number Generator

```
def fib():  
    """ returns a generator for  
        all Fibonacci numbers """  
    a, b = 1, 1  
    yield a  
    while True:  
        yield b  
        a, b = b, a+b
```

```
>>> g = fib()  
>>> g  
<generator object fib at 0x1704fa8>  
  
>>> next(g)  
1  
>>> next(g)  
1  
>>> next(g)  
2  
>>> [next(g) for i in range(10)]  
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Example2: "Moving Average"

```
def avg():
    """ compute average of "online" inputs """
    s = 0
    cnt = 0
    while True:
        tp = int(input("What's the temperature today? "))
        cnt+=1
        s+=tp
        print("Average temperature so far:")
        yield s/cnt
```

```
>>> m = avg()
>>> type(m)
<class 'generator'>
```

```
>>> next(m)
What's the temperature today? 25
Average temperature so far:
25.0
>>> next(m)
What's the temperature today? 26
Average temperature so far:
25.5
>>> next(m)
What's the temperature today? 20
Average temperature so far:
23.666666666666668
```

Example 3: Merging Sorted, Infinite Generators

- Suppose `gen1` and `gen2` are generators of **sorted** streams, **both infinite**.
- We wish to produce a new sorted stream which is the merge of both.

```
def merge(gen1, gen2):
    """ on input gen1, gen2,
        two generators of infinite sorted streams,
        produces the sorted merge of the two """

    left = next(gen1)
    right = next(gen2)
    while True:
        if left <= right:
            yield left
            left = next(gen1)
        else:
            yield right
            right = next(gen2)
```

Example 3: Merging Sorted, Infinite Generators (cont.)

```
>>> nat1 = naturals()
>>> nat2 = naturals()
>>> nat_twice = merge(nat1, nat2)

>>> next(nat_twice)
0
>>> next(nat_twice)
0
>>> next(nat_twice)
1
>>> next(nat_twice)
1
>>> [next(nat_twice) for i in range(10)]
[2, 2, 3, 3, 4, 4, 5, 5, 6, 6]
```

- Would `merge` work properly for **finite generators**?

Finite Generators

- When a finite generator is exhausted, a `StopIteration` runtime error is raised.

```
def f1(n):  
    """ finite generator """  
    for i in range(n):  
        yield i
```

```
>>> g1 = f1(2)  
>>> next(g1)  
0  
>>> next(g1)  
1  
>>> next(g1)  
Traceback (most recent call last):  
      next(g1)  
StopIteration
```

```
def f2():  
    """ finite generator """  
    yield "a"  
    yield "a"  
    yield "b"
```

```
>>> g2 = f2()  
>>> next(g2)  
'a'  
>>> next(g2)  
'a'  
>>> next(g2)  
'b'  
>>> next(g2)  
Traceback (most recent call last):  
      next(g2)  
StopIteration
```

Finite Generators and Generator Expressions

- Finite generators can also be created using **generator expressions** – same syntax as list comprehension but inside () instead of []:

```
>>> n = 3
>>> g3 = (x**2 for x in range(n)) # note the () instead of []
>>> type(g3)
<class 'generator'>
>>> next(g3)
0
>>> next(g3)
1
>>> next(g3)
4
>>> next(g3)
Traceback (most recent call last):
  next(g3)
StopIteration
```

More on Finite Generators

- We remark that it would **not be possible** to handle even finite but **very large** collections without the **lazy evaluation** mechanism.
- For example, representing the first 10^8 integers can fit in just under **1GB RAM**. But what about 2^{100} elements?
A collection with 2^{100} elements will not fit in Amazon, Google, and NASA computers, even if taken together.

DON'T TRY THIS AT HOME (well, really do try it once...)

typing this will cause IDLE to get stuck

```
>>> mylist = [n for n in range(2**100)]
```

- As we saw, we can represent this huge collection easily using generators (at no time the entire collection will be stored in memory)

An Attempt to Merge Sorted, **Finite** Generators

- Should the generators in `merge` really be infinite?

```
def f(n):  
    """ a finite generator """  
    for i in range(n):  
        yield i
```

```
>>> nat = naturals()  
>>> zero_one = f(2)  
>>> res = merge(nat, zero_one)  
>>> next(res)  
0  
>>> next(res)  
0  
>>> next(res)  
1  
>>> next(res)  
1  
>>> next(res)  
Traceback (most recent call last):  
  right = next(gen2)  
StopIteration
```

What went **wrong**?

The merged generator, `res`, was not yet exhausted, however one of the arguments to `merge`, `zero_one` was exhausted. The merging procedure still invoked `next(gen2)`. This has caused a `StopIteration` error.

Example 4: A Permutations Generator

- The following generator produces all **permutations** of a given (finite) set of elements.

```
def permutations(seq):  
    if len(seq) <= 1:  
        yield seq  
    else:  
        for perm in permutations(seq[1:]): # all except 1st  
            for i in range(len(seq)): # location to insert 1st  
                yield perm[:i] + seq[0:1] + perm[i:]
```

- The elements should be given as an indexable sequence (e.g. list, tuple, or string)
- It allows one to produce all permutations, one by one, without generating or storing all of them at the same time.

Example 4: A Permutations Generator (cont.)

```
>>> a = permutations("mit")
>>> list(a)
['mit', 'imt', 'itm', 'mti', 'tmi', 'tim']
```

Make sure the size
is under control

```
>>> a = permutations("") # the empty string
>>> list(a)
['']
```

Limitations of Infinite Generators

- Given a possibly infinite iterator:
 - Create a generator that generates the **reverse** sequence.

Can't be done! Why?

- Create a generator that generates only the elements that appear **more than once** in the original sequence.

Can't be done! Why?

Idea: remember previously seen elements.

Practical limitation: memory “explosion”

Fundamental limitation: When there are no more repetitions, a `next` command will never terminate...

Limitations of Infinite Generators

More generally, the property that we need is "finite delay":

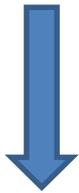
- The time it takes to generate the next single item of the generated sequence is **finite**.
 - One may also talk about **polynomial** delay, **linear** delay, **constant** delay, etc.
-
- What delay did we have in all the previous examples?

Iterators

- **Iterators** in Python provide access to a **given collection** of items one by one (this is in contrast to generators, which do not have to relate to any given collection)
- Each type of collection in Python has its own **iterator type**
 - For example `list_Iterator`, `str_Iterator`, `dict_iterator`, etc.
- When using a **for** loop to iterate over a collection, **Behind the scenes** a suitable **iterator** object is created first, and then the collection's items are extracted, one by one, using `next()`.
- Classes that enable **for** loops are called **iterables**.
 - This includes any class with an `__iter__` method, which is responsible for creating a suitable iterator

For loop: Behind the Scenes

```
>>> for e in iterable:  
    do_something_with(e)
```



1. `it` ← an iterator for `iterable`
2. While True:
 - 2.1 `e` ← `next(it)`
 - 2.2 `do_something_with(e)`
3. Handle `StopIteration` by ignoring it

Handling Exceptions in Python (for reference only)

A short diversion:

Handling Errors with **try** and **except**

- Python provides an elaborate mechanism to handle run time errors. For example, division by zero causes a `ZeroDivisionError` .

```
>>> 5/0
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#37>", line 1, in <module>
```

```
    5/0
```

```
ZeroDivisionError: int division or modulo by zero
```

- Such errors disrupt the flow of control in a program execution. We may want to **detect** such error and allow the flow of control to **continue**.
- This may not be so important in the small programs written in this course, but becomes meaningful in large software projects.
- Python enables such detection, using the keywords **try** and **except**.

Handling Errors: **try** and **except**: example

```
def division(a,b):  
    try:  
        return a/b  
    except ZeroDivisionError:  
        print("division by zero")
```

- Let us now apply this function in two different cases:

```
>>> division(5,6)  
0.8333333333333334
```

```
>>> division(5,0)  
division by zero
```

- We will employ this error handling mechanism to enable merging any non-empty sorted iterators, finite or infinite .

More on `try` and `except`

- The example in the previous slide is not so good – we can solve this problem with an `if` statement.
- However, consider a situation where we would need to write many `if` statements to handle division by zero. Instead, `try/except` wrap the whole block:

```
def compute(...):  
    try  
        # a long computation, with several steps  
        # that may cause zero division  
  
    except ZeroDivisionError:  
        print("division by zero")
```

- We will also use `try/except` when it is either **impossible** or **expensive** to check for the condition in advance. Example – when `we invert a matrix`, checking in advance that it is not singular `would take as much time` as inverting, so it makes more sense to try to invert, and raise an **exception** if we discover that the matrix is singular while we do it.
- We can have multiple `except` clauses; a list of exceptions to be handled in each clause; and the last clause may omit exception names (to handle all others)

Back to Merging **Any** Non-Empty, Sorted iterators

```
def merge2(iter1, iter2):  
    """ on input iter1, iter2, two non-empty sorted iterators, not  
        necessarily infinite, produces sorted merge of the two iterators """  
    left = next(iter1)  
    right = next(iter2)  
    while True:  
        if left < right:  
            yield left  
            try:  
                left = next(iter1)  
            except StopIteration:          # iter1 is exhausted  
                yield right  
                remaining = iter2  
                break
```

merge2 : cont.

```
else:
    yield right
    try:
        right = next(iter2)
    except StopIteration:      # iter2 is exhausted
        yield left
        remaining = iter1
        break
# end of the while loop
for elem in remaining:      # protects against StopIteration
    yield elem
```

Merge2: Examples of Executions

```
>>> iter1 = (x**2 for x in range(4))
>>> iter2 = natural()
>>> merged = merge3(iter1,iter2)
>>> [next(merged) for i in range(14)]
[0, 1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10]
```

```
>>> iter1 = (x**2 for x in range(5))
>>> iter2 = (x**3 for x in range(6))
>>> merged = merge3(iter1,iter2)
>>> [next(merged) for i in range(11)]
[0, 0, 1, 1, 4, 8, 9, 16, 27, 64, 125]
```

Finally, let's see what happens when the original iterators/generators are not sorted:

```
>>> iter1 = ((-1)**x*x**2 for x in range(5))
>>> iter2 = (x**3 for x in range(6))
>>> merged = merge3(iter1,iter2)
>>> [next(merged) for i in range(11)]
[0, 0, -1, 1, 4, -9, 8, 16, 27, 64, 125]
# garbage in, garbage out
```

Iterators in Python (for reference only)

Iterators - Motivation

- **Linked lists** and Python's built-in **lists** (arrays) are two ways to represent a collection of elements. There are others, such as **trees**, **hash tables**, and more.
- It is **desirable** that functions that use the data as part of a computation should be as oblivious as possible to such **internal representation**, which **may change over time**.
- This holds in particular for **iterating over** collections of elements
 - This idea is captured in a concrete way by Python's **iterators**. Iterators will provide a **generic access** to a collection of items one by one.

Iterables

- An **iterable** object is an object capable of returning its members one at a time.
- In particular, we can use a **for loop** on iterables
- Examples of **iterables** include:
 - all **sequence** types such as **list**, **str**, **tuple** and **range**
 - some **non-sequence** types such as **dict**, **set** (and also **File**)
 - objects of any user defined classes with an **__iter__** or **__getitem__** method (this is how you make your new class iterable. But we will not see this)

(see <http://docs.python.org/dev/glossary.html#term-iterable>)

Iterables

- For example, `range` in Python is `iterable`:

```
>>> a = range(10)
```

```
>>> type(a)
<class 'range'>
```

```
>>> a
range(0, 10)
```

```
>>> for i in a:
    print(i)
```

```
0
```

```
1
```

```
2
```

```
...
```

Iterators

- An **iterator** is an object representing a stream of data.
- Each **iterable** type in Python has its own corresponding **iterator** type, created using the built-in `iter()` function.
- Repeated calls to the built-in function `next(it)`, where `it` is an iterator (or calls to the iterator's `__next__()` method) return successive items in the stream.
- When no more data are available a **StopIteration exception** is raised instead. At this point, the iterator object is "exhausted", and any further calls to `next(it)` just raise **StopIteration exception** again.

```
>>> it = iter([0,1,2])
>>> type(it)
<class 'list_iterator'>
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
```

```
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    next(it)
StopIteration
```

```
>>> it = iter("ab")
>>> type(it)
<class 'str_iterator'>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
```

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
StopIteration
```

Iterables and Iterators

- We can create an iterator by calling the function `iter` with an **iterable object** argument (like list, tuple, str, dict, range , etc.)
- This function does not modify the **original** iterable object.

```
>>> table = {"Benny":72,"Daniel":82,"Amir":92}
```

```
>>> next(table)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#13>", line 1, in <module>
```

```
    next(table)
```

```
TypeError: dict object is not an iterator
```

```
>>> it = iter(table)
```

```
>>> next(it)
```

```
'Amir'
```

```
>>> next(it)
```

```
'Benny'
```

```
>>> next(it)
```

```
'Daniel'
```

```
>>> next(it)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#18>", line 1, in <module>
```

```
    next(it)
```

```
StopIteration
```

For loop

- We mentioned that a **for** loop over an iterable actually uses an iterator.
- Here are the details:

```
>>> elems = ['a','b','c']  
>>> for e in elems:  
    print(e)
```

```
a  
b  
c
```

=

```
>>> elems = ['a','b','c']  
It = iter(elems)  
while True:  
    try:  
        print(next(it))  
    except StopIteration:  
        break
```

```
a  
b  
c
```

Iterators and **for** Loops

- When we loop over an iterable using **for**, an **iterator** is created first, and then the items are returned, one by one, using `next()`.

```
>>> table = {"Benny":72,"Daniel":82,"Amir":92}
>>> for key in table:           # an iterator is created
    print(key)                 # "under the hood"
                                # more details later
```

```
Amir
Benny
Daniel
```

- As we see from this example, a dictionary (when transformed into an iterator), returns the keys one by one.
- Files return the lines one by one, etc.

Iterators have "states"

- We can turn an iterator into a list as well. This list will reflect the **current state** of the iterator, **not** its original state:

```
>>> table = {"Benny":72,"Daniel":82,"Amir":92}
```

```
>>> it = iter(table)
```

```
>>> next(it)
```

```
'Amir'
```

```
>>> list(it)
```

```
['Benny', 'Daniel']
```

```
>>> next(it)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#82>", line 1, in <module>
```

```
next(it)
```

```
StopIteration
```

Thou Shalt Not Modify an iterable during Iteration

- If we add or remove elements from an iterable during iteration, **strange things** may happen. For example

```
>>> elems = ['a', 'b', 'c']
>>> for e in elems:
    print(e)
    elems.remove(e)

a
c

>>> elems
['b']
```

adapted from

<http://unspecified.wordpress.com/2009/02/12/thou-shalt-not-modify-a-list-during-iteration/>

Iterators as a Tool for **Abstraction**

- The use of iterators **hides the implementation** of data collections. For example, when we see the code

```
for x in SomeCollection:
```

```
....
```

We do not know if SomeCollection is a list, a dict, or any user defined data collection. We just want to get our hands on its elements.

- Defining an **iterator** for SomeCollection will solve this problem, allowing us to use a for loop regardless of the actual implementation.
- Furthermore, we can later modify the implementation of SomeCollection, for example change it from a list to a dict, and the code using it (for loop) will not have to be changed.
- Similarly when we use **next(it)**, **it** may be an iterator of any type of a data collection, with any order of traversal.