Extended Introduction to Computer Science CS1001.py

Lecture 17A: Finite and Infinite Iterators

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

Founding TA and Python Guru: Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

HW 4, Question 8
(the Joy of Floating Point Arithmetic)

```
>>> 2.0**52 == 2.0**52+1.0
False
>>> 2.0**53 == 2.0**53+1.0
True
```

This is easy to explain, given our understanding of how floating point numbers are represented, and the fact that the fraction part is exactly 52 bit long.

But what about

```
>>> 2.0**53 == 2**53+1
False
```

Here things seem somewhat more mysterious…

HW 4, Question 8
the Joy of Floating Point Arithmetic

```
>>> 2.0**53 == 2**53+1
False
```

To perform the comparison, the computer has to cast either the integer into a float (which is the "standard"), or the float into an integer. It seems that when large numbers are involved, the latter takes place. See the part on big-int in the function make_compare_fun (line 90 and onwards) of PyPy floatobj.py code (not the most enjoyable reading, mind you).

https://bitbucket.org/pypy/pypy/src/789fb549e0afde4710aa97497c424599cd36180f/pypy/objspace/std/floatobject.py?at=default&fileviewer=file-view-default

Thanks to Omer Chor for figuring this out!

Lectures 15-18 Highlights

Data Structures

1. Python's lists (arrays) vs. linked list

2. binary search trees

3. hash tables
   - The dictionary problem (find, insert, delete).
   - hash functions, Python's hash

4. TODAY - iterators

Lecture 19 part A, Plan

- Iterators
  - Lazy (delayed) evaluation.
  - Infinite iterators.
  - Examples: Merging sorted iterators.
  - Handling Errors: try and except

Iterators and Generators

- Linked lists and Python's built-in lists (arrays in other programming languages) are two ways to represent a collection of elements. There are other ways, such as trees, dictionaries, and more.

- It is desirable that functions, which use the data as part of a computation, will be as oblivious as possible to such internal representation, which may change over time.
  - This general idea is captured in a concrete way by Python's iterators.
  - Iterators provide a generic access to a ordered collection of items. So generic that it will even allow us to access an infinite collection (also known as stream)!
  - Python's generators are tools to create iterators .

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 like dict, set and 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 get into this here)

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

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)
  - certain non-sequence types like dict, set and 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 get into this here).

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

range is a special iterable class.

```
>>> a=range(10)
>>> type(a)
<class 'range'>
>>> a
range(0, 10)
>>> a[2]
2
```

Thou Shalt Not Modify an iterable during Iteration

- If we add or remove elements to/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

- An iterator is an object representing a stream of data.
- Each iterable type in Python has its own iterator type, created using the built-in iter()
- 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])
>>> 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
```

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. In fact, when we loop over an iterable using for, an iterator is created first, and then the items are called, one by one, using next() .

```
>>> table = {"benny“:72,"rani“:82,"raanan“: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)
'rani'
>>> next(it)
'benny'
```

Iterables and Iterators, cont.

```
>>> next(it)
'raanan'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(it)
StopIteration

>>> table = {"benny“:72,"rani“:82,"raanan“:92}
>>> for key in table:             # an iterator is created
      print(key)                  # “under the hood”
                                  # more details later
rani
benny
raanan
```

Iterables and Iterators, cont.

- 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.

- 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,"rani“:82,"raanan“:92}
>>> it = iter(table)
>>> next(it)
'rani'
>>> list(it)
['benny', 'raanan']
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    next(it)
StopIteration
```

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 have to know if SomeCollection is a list, a tuple, a string, a dict, or any user defined data collection. Furthemore, we can later modify the implementation of SomeCollection, for example change it from a list to a dict, and the code using it (called client code) will not have to be changed.

- Similarly when we use next(it), it may be an iterator of any kind of a data collection, with any order of traversal.

Iterables, Iterators, and Generators: More Examples

```
>>> mylist = [x for x in range(10**8)]
>>> it1 = iter(mylist)
>>> it2 = (x for x in range(10**8))  #  note the () instead of []

>>> type(mylist)
<class 'list'>
>>> type(it1)
<class 'list_iterator'>
>>> type(it2)
<class 'generator'>     #list comprehension syntax inside () is one way to
                        #create a generator. The other will be shown next

>>> # mylist
        # typing this without the comment will clobber your screen
        # and most likely will cause your Python shell to crash
>>> it1
<list_iterator object at 0x17027d0>
>>> it2
<generator object <genexpr> at 0x1704f08>
```

Iterables, Iterators, and Generators, cont.

```
>>> len(it2)
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    len(it2)
TypeError: object of type 'generator' has no len()
```

- it1 and it2 are iterators (respectively generator) representing the first 10^8 integers.
- These integers can fit in just under 1GB RAM. But, can we have iterators representing even more items?

```
>>> it3 = (x for x in range(2**100))  # again, note the ()
>>> next(it3)
0
>>> next(it3)
1
```

- An iterable with 2^{100} elements will not fit in Amazon, Google, NSA, and NASA computers, even if taken together.
- Iterators and generators represent streams, but produce only one element at a time. Therefore, there is no problem representing a 2^{100} long stream!

Generators for Infinite Streams

In fact, there is no problem representing streams with countably many elements.

To do that, we will introduce generator functions.

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 does not refer to a global variable, which may have changed). This is now going to change .

```
def naturals():
    """ a generator for all natural numbers """
    n=1
    while True:
        yield n
        n+=1
```

Generators for Infinite Streams, cont.

A function that contains a yield statement is termed a generator function. When a generator function is called, the actual arguments are bound to the function's formal argument names in the usual way, but no code in the body of the function is executed. Instead, a generator—iterator object is returned.

```
>>> naturals()
<generator object natural at 0x16f60d0>
>>> nat = naturals()
>>> nat
<generator object natural at 0x16f60a8
```

Generators, cont.

- nat is a generator--iterator. To get its "returned value", which is specified by the yield statement, we invoke next .

```
>>> next(nat)
1
>>> next(nat)
2
>>> [next(nat) for i in range(10)]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

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

```
>>> nat2 = naturals()
>>> next(nat2)
1
>>> next(nat)
13
```

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, and also avoids repeated evaluations by memoization (caching).

- The "opposite'' of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is the “standard” evaluation behavior, used in most programming languages.

Lazy Evaluation (cont.)

- Python's iterators and generators employ lazy evaluation.
- The next item is evaluated only when it is required, by means of executing next(). We remark that it would not be possible to handle finite but very large iterators/generators, or infinite iterators/generators, without the lazy evaluation mechanism.

- Scheme, the good old programming language used in TAU (and elsewhere) has a special syntax, enabling the delay and force of an evaluation of an expression. These make possible in Scheme the use of very large streams, and of infinite streams.

Eager evaluation

We have been using eager evaluation all the time and did not know it!

Just like Le Bourgeois Gentilhomme, by Molière:

“Good heavens! For more than forty years I have been speaking prose without knowing it.”

מסייה ז'ורדן, בקומדיה "גם הוא באצילים" מאת מולייר:

"חיי ראשי! זה למעלה מארבעים שנה שאני מדבר פרוזה ואינני יודע זאת כלל".

(Quotations taken from Wikipedia)

A Fibonacci Numbers' Generator

```
def fib():
    """ a generator for all Fibonacci numbers"""
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a+b

>>> Fib = fib()
>>> Fib
<generator object fib at 0x1704fa8>
```

- Again, Fib is a generator--iterator, so to get its “returned value'', which is specified by the yield statement, we invoke next() .

```
>>> next(Fib)
1
>>> next(Fib)
1
>>> next(Fib)
2
>>> [next(Fib) for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Execution Specification

When a yield statement is encountered,

yield expression_list

the state of the function is "frozen", and the value of expression_list 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, 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/)

This is similar to what is called co-routine, in contrast with a "normal function" call which is also termed sub-routine.

Merging Sorted, Infinite Iterators

Suppose iter1 and iter2 are sorted iterators, and in addition, both are infinite. We wish to produce a new sorted iterator, which is the merge of both.

```
def merge(iter1,iter2):
    """ on input iter1, iter2, two infinite orted iterators,
    produces the sorted merge of the two iterators ""“

    left = next(iter1)
    right = next(iter2)
    while True:
        if left<right:
            yield left
            left = next(iter1)
        else:
            yield right
            right = next(iter2)
```

Merging Sorted, Infinite iterators: Execution

```
>>> nat1=natural()
>>> nat2=natural()
>>> nat3=merge(nat1,nat2)
```

nat3, too, is a generator--iterator, so to get its “returned value'', which is specified by the yield statement, we invoke next.

```
>>> next(nat3)
1
>>> next(nat3)
1
>>> next(nat3)
2
>>> next(nat3)
2
>>> [next(nat3) for i in range(10)]
[3, 3, 4, 4, 5, 5, 6, 6, 7, 7]
```

An Attempt to Merge Sorted, Finite iterators

Should the iterators in merge really be infinite ?

```
>>> nat1 = natural()
>>> nat2 = (n-2 for n in range(3))
>>> nat3 = merge(nat1,nat2)
>>> next(nat3)
-2
>>> next(nat3)
-1
>>> next(nat3)
0
>>> next(nat3)
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    next(nat3)
  File "/Users/benny/Documents/InttroCS2011/Code/intro17/lecture17.py", line 30, in merge
    right=next(iter2)
StopIteration
```

What went wrong?

The merged iterator, nat3, was not yet exhausted, yet one of the arguments to merge, nat2, was exhausted. The merging procedure is unaware of this, and still invoked next(iter2) , causing a StopIteration error.

Handling Errors: 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 rather 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

We could also solve the division by zero problem using one if statement. The following example shows a situation where we would need to write many if statements, so try/except is better

```
def compute(…):
    try
        # a long computation, with several steps
        # that may cause  zero divide
    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 could 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).

For loop

We mentioned that a for loop over an iterable using for, actually uses an iterator. We now show an example:

```
>>> elems = ['a','b','c‘]
```

```
for e in elems:
    print(e)
```

```
a
b
c
```

Is the same as

```
It = iter(elems)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

```
a
b
c
```

Merging Non-Empty, Sorted iterators. Take 2

```
def merge3(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
# *
# *   continued on next page
```

merge3 : 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)
```

Merge3: 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, lets 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
```