# INF 212
# ANALYSIS OF PROG. LANGS
# *ITERATORS, GENERATORS AND COROUTINES*

Instructors: Crista Lopes

# Iterators

# Iterators, Generators

# Iterators, Generators, Coroutines

# Iterators

# Iterators – what is the problem?

- Problem: accidentally large intermediary lists

- Example, we want to printout:
  - a stream of positive random numbers $n < 1$ where
  - $abs(n_{i+1} - n_i) > 0.4$
  - stream stops when $n < 0.1$

How would you do this?

# Iterators – 1st attempt

```
import random
def randomwalk_list():
    last, rand = 1, random.random() # init candidate elements
    nums = []                       # empty list
    while rand > 0.1:               # threshold terminator
        if abs(last-rand) >= 0.4:   # accept the number
            last = rand
            nums.append(rand)       # add latest candidate to nums
        else:
            print '*',              # display the rejection
        rand = random.random()      # new candidate
    nums.append(rand)               # add the final small element
    return nums
```

```
for num in randomwalk_list():
    print num,
```

We need to generate the entire list before printing any number out!

# Iterators – 2nd attempt

"static" function-local variable

```
import random
def randomwalk_static(last=[1]):    # init the "static" var(s)
    rand = random.random()          # init a candidate value
    if last[0] < 0.1:               # threshhold terminator
        return None                 # end-of-stream flag
    while abs(last[0]-rand) < 0.4:  # look for usable candidate
        print '*',                  # candidate's existence
        rand = random.random()      # new candidate
    last[0] = rand                  # update the "static" var
    return rand
```

Better, but clumsy

```
num = randomwalk_static()
while num is not None:
    print num,
    num = randomwalk_static()
```

# Iterators

```
import random
class randomwalk_iter:
    def __init__(self):
        self.last = 1                 # init the prior value
        self.rand = random.random() # init a candidate value
    def __iter__(self):
        return self                   # simplest iterator creation
    def next(self):
        if self.rand < 0.1:           # threshhold terminator
            raise StopIteration       # end of iteration
        else:                         # look for usable candidate
            while abs(self.last-self.rand) < 0.4:
                print '*',            # candidate's existence
                self.rand = random.random() # new candidate
            self.last = self.rand     # update prior value
            return self.rand
```

A little verbose here!

Problem solved here!

```
for num in randomwalk_iter():
    print num,
```

# What are iterators, really?

- Objects that keep state for traversing an abstract collection
- Closures that get passed around in every next()

- btw, objects and closures are related...

# Same iterator in Java

```
import java.util.Iterator;
class IterExample implements Iterator<Double> {
    private double last = 1;
    private double rand = Math.random();

    public boolean hasNext() {
       return (rand >= 0.1);
    }
    public Double next() {
       if (rand >= 0.1) {
           while (Math.abs(last - rand) < 0.4) {
              System.out.print("* ");
              rand = Math.random();
           }
           last = rand;
       }
       return rand;
    }
    public void remove() { }
```

# Same iterator in Java

```
    // ...continued
    public static void main(String[] args) {
       IterExample it = new IterExample();
       while (it.hasNext())
           System.out.print(it.next() + " " );
    }
}
```

# Generators

# Generators

- Generators are functions that "yield" values every time they are called

```
def gen123():
    yield 1
    yield 2
    yield 3
```

# Generator

```
import random
def randomwalk_gen():
    last = 1                          # initialize candidate elements
    rand = random.random()            # initialize candidate elements
    while rand > 0.1:                 # threshhold terminator
         print'*',                    # candidate's existence
         if abs(last-rand) >= 0.4: # accept the number
            last = rand               # update prior value
            yield rand                # return AT THIS POINT
         rand = random.random()       # new candidate
    yield rand
```

Nice here too!

```
for num in randomwalk_gen():
    print num,
```

# Same generator in C#

```
using System;
using System.Collections.Generic;

namespace GenExample {
    class Program {
        static Random random = new Random();

        static IEnumerable<double> RandomWalkGen() {
            double last = 1;
            double rand = random.NextDouble();
            while (rand > 0.1) {
                Console.Write("* ");
                if (Math.Abs(last - rand) >= 0.4)
                {
                    last = rand;
                    yield return rand;
                }
                rand = random.NextDouble();
            }
            yield return rand;
        }
```

# Same generator in C#

```
        // ... continued

         static void Main(string[] args)
         {
             foreach (double d in RandomWalkGen())
                 Console.Write(d + " ");
         }
    }
}
```

# Generators – the magic “yield”

- http://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-generators-coroutines/
- Key idea: toggle between 2 continuations: one in the outter code and one in the generator

# Generators

- Java: no equivalent
  - must use iterators
- C++: no equivalent
  - But boost library supports them via coroutines

- All other [major] languages have support for them

# Coroutines

# Coroutines

- Procedures/functions that allow multiple entry points
  - They 'remember' the last state of their execution
  - They call on each other as peers rather than caller/callee

- Appropriate scenario:
  - A function that produces a stream of data
  - A function that consumes a stream of data
  - Which one calls which?

# Motivating example

Decompression followed by parsing

```
/* Decompression code */
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                emit(c);
        } else
            emit(c);
    }
    emit(EOF);
```

```
/* Parser code */
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = getchar();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
```

http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

# Motivating example

Decompression followed by parsing – **option 1: parser calls decompressor**

```
int decompressor(void) {
    static int repchar;
    static int replen;
    if (replen > 0) {
        replen--;
        return repchar;
    }
    c = getchar();
    if (c == EOF)
        return EOF;
    if (c == 0xFF) {
        replen = getchar();
        repchar = getchar();
        replen--;
        return repchar;
    } else
        return c;
}
```

```
/* Parser code */
    while (1) {
        c = decompressor();
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = decompressor();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
```

# Motivating example

Decompression followed by parsing – **option 2: decompressor calls parser**

```
/* Decompression code */
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                parser(c);
        } else
            parser(c);
    }
    parser(EOF);
```

```
 void parser(int c) {
    static enum {
        START, IN_WORD
    } state;
    switch (state) {
        case IN_WORD:
        if (isalpha(c)) {
            add_to_token(c);
            return;
        }
        got_token(WORD);
        state = START;
        /* fall through */
        case START:
        add_to_token(c);
        if (isalpha(c)) state=IN_WORD;
        else got_token(PUNCT);
        break;
    }
}
```

# Motivating example

Decompression followed by parsing – **option 3: "cooperative partners"**

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
               crReturn(c);
        } else
           crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

```
void parser(int c) {
    crBegin;
    while (1) {
      * first char already in c */
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
             add_to_token(c);
            crReturn( );
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
       crReturn( );
    }
    crFinish;
}
```

# Motivating example

- Dirty little secrets of this code:
  - Hackery needed because C doesn't want to do coroutines
  - crBegin, crFinish, crReturn are HORRIBLE macros that I don't dare to show

# Coroutines

- Not just pairs of functions, but any number of functions
- Functions can specify which other function to yield to
- Implementation: stack per coroutine, continuations
- Lightweight alternative to threads
  - No real concurrency, just switching functions
  - Very nice model for processing data streams
- Fell out of favor in the 80s
  - May result in spaghetti code
  - May see a come back