Discussion 10:

Iterators, Generators and Streams

Nancy Shaw (nshaw99@berkeley.edu)
Caroline Lemieux (clemieux@berkeley.edu)
April 18th, 2019

Iterators and Generators

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

ERROR

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

ERROR

```
next(i)
```

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

ERROR

```
next(i)
```

“a”

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

ERROR

```
next(i)
```

“a”

```
next(i)
```

Iterators vs. Iterables

```
s = “am a string”
i = iter(s)
```

str is an iterable

get its iterator with iter()

```
next(s)
```

ERROR

```
next(i)
```

“a”

```
next(i)
```

“m”

How to go through iterables without calling next

`for i in iterable` goes through all the things in iterable, by calling `next`

Calling `list(iterable)` makes a list of all the things we get by calling `next`

Do 1.1

Pausing vs. stopping a video

Kind of like `yield`

Kind of like `return`

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i
```

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i

>>> generate_up_to(5)
<generator object ...>
```

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i

>>> generate_up_to(5)
<generator object ...>
```

When python sees a **`yield`** in a function, calling that function returns a *generator*

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i

>>> generate_up_to(5)
<generator object ...>
>>> g = generate_up_to(5)
>>> next(g)
```

When python sees a **`yield`** in a function, calling that function returns a *generator*

Calling next on the generator “plays” that function, until yield, where it “pauses”

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i

>>> generate_up_to(5)
<generator object ...>
>>> g = generate_up_to(5)
>>> next(g)
0
```

When python sees a **`yield`** in a function, calling that function returns a *generator*

Calling next on the generator “plays” that function, until yield, where it “pauses”

Generators

```
def generate_up_to(n):
    for i in range(0, n):
        yield i

>>> generate_up_to(5)
<generator object ...>
>>> g = generate_up_to(5)
>>> next(g)
0
>>> next(g)
1
```

When python sees a **`yield`** in a function, calling that function returns a *generator*

Calling next on the generator “plays” that function, until yield, where it “pauses”

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
```

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
0
```

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
0
```

Calling a regular functions “plays” that function, until return, where it “stops”

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
0
>>> generate_up_to(5)
```

Calling a regular functions “plays” that function, until return, where it “stops”

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
0
>>> generate_up_to(5)
0
```

Calling a regular functions “plays” that function, until return, where it “stops”

Concept check: yield vs return

```
def generate_up_to(n):
    for i in range(0, n):
        return i

>>> generate_up_to(5)
0
>>> generate_up_to(5)
0
```

Calling a regular functions “plays” that function, until return, where it “stops”

When we call it again, it “plays” from the start

Yield from

```
def generate_up_to(n):
    for i in range(0, n):
        yield i
```

Same thing!

```
def generate_up_to(n):
    yield from range(0,n)
```

Recursive generator

```
def generate_down_to_zero(n):
    if n == 0:
        yield 0
    else:
        yield n
        yield from generate_down_to_zero(n-1)
```

Do 1.1 (the other 1.1)

Attendance

links.cs61a.org/caro-disc

next(cats)

Streams (back to scheme)

An infinite natural number generator in Python

(demo)

An infinite natural number generator… in scheme?

(demo)

What’s a stream:

A “lazy” scheme list

- Lazy because it evaluates its first element….
- … but then is lazy and doesn’t evaluate the second

How do I make a stream?

```
(cons-stream <operand1> <operand2>)
```

Another special form!

1. Evaluate operand1 to get val1
2. Construct promise containing operand2
3. Return a pair (val1, promise of operand2)

How do I make a stream?

```
(cons-stream <operand1> <operand2>)
```

Another special form!

1. Evaluate operand1 to get val1
2. Construct promise containing operand2
3. Return a pair (val1, promise of operand2)

(demo)

How do I make a stream?

```
(cons-stream <operand1> <operand2>)
```

Another special form!

1. Evaluate operand1 to get val1
2. Construct promise containing operand2
3. Return a pair (val1, promise of operand2)

Need special `(cdr-stream s)` to get the cdr properly

Important: `cdr-stream` evalutes its value **once**, then saves that for later calls

How do I make a stream?

```
(cons-stream <operand1> <operand2>)
```

Another special form!

1. Evaluate operand1 to get val1
2. Construct promise containing operand2
3. Return a pair (val1, promise of operand2)

Need special `(cdr-stream s)` to get the cdr properly

Important: `cdr-stream` evalutes its value **once**, then saves that for later calls

(demo)

Stream Recap

1. nil is the empty stream
2. cons-stream constructs a stream
3. car gets the first element of the stream
4. cdr-stream computes and returns the rest of the stream (it only computes once, and saves the value)
    a. Promise is “forced” if we’ve computed its value