

Iterators

Announcements

Iterators

Iterators

A container can provide an iterator that provides access to its elements in order

iter(iterable): Return an iterator over the elements of an iterable value

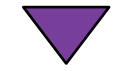
next(iterator): Return the next element in an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> w = u
>>> next(w)
4
```

(Demo)

Discussion Question

What will be printed?



```
a = [1, 2, 3]
b = [a, 4]
c = iter(a)
d = c
print(next(c))
print(next(d))
print(b)
```

Discussion Question

What will be printed?

▼
a = [1, 2, 3]
b = [a, 4]
c = iter(a)
d = c
print(next(c))
print(next(d))
print(b)

1
2
[[1, 2, 3], 4]

Map Function

Map

`map(func, iterable)`: Make an iterator over the return values of calling `func` on each element of the `iterable`.

(Demo)

Built-in Functions for Iteration

Many built-in Python sequence operations return iterators that compute results lazily

<code>map(func, iterable):</code>	Iterate over <code>func(x)</code> for <code>x</code> in <code>iterable</code>
<code>filter(func, iterable):</code>	Iterate over <code>x</code> in <code>iterable</code> if <code>func(x)</code>
<code>zip(first_iter, second_iter):</code>	Iterate over co-indexed <code>(x, y)</code> pairs
<code>reversed(sequence):</code>	Iterate over <code>x</code> in a sequence in reverse order

Example: summation

```
def cube(k):  
    return pow(k, 3)
```

```
def summation(n, term):  
    """Sum the first n terms of a sequence.
```

```
>>> summation(5, cube)
```

```
225  
-----  
0 + 1 + 8 + 27 + 64 + 125
```

```
total, k = 0, 1
```

```
while k <= n:
```

```
    total = total + term(k)
```

```
    k = k + 1
```

```
return total
```

Re-implement summation:

```
def summation_list_comprehension(n, term):
```

```
    return
```

Include [`<map exp>` `for <name> in <iter exp>`]

```
def summation_map(n, term):
```

```
    return
```

Include `map(func, iterable)`

Example: summation

```
def cube(k):  
    return pow(k, 3)
```

```
def summation(n, term):  
    """Sum the first n terms of a sequence.
```

```
>>> summation(5, cube)
```

```
225  
-----
```

0 + 1 + 8 + 27 + 64 + 125

```
total, k = 0, 1
```

```
while k <= n:
```

```
    total = total + term(k)
```

```
    k = k + 1
```

```
return total
```

Re-implement summation:

```
def summation_list_comprehension(n, term):  
    return sum([term(k) for k in range(1, n + 1)])
```

Include [`<map exp>` `for <name> in <iter exp>`]

```
def summation_map(n, term):
```

```
    return sum(map(term, range(1, n + 1)))
```

Include `map(func, iterable)`

List comprehension vs. map function

```
def cube(k):  
    print("cubing", k)  
    return pow(k, 3)
```

What will be printed for each?

```
>>> x = [cube(k) for k in range(1, 6)]
```

```
cubing 1  
cubing 2  
cubing 3  
cubing 4  
cubing 5
```

```
>>> x  
[1, 8, 27, 64, 125]
```

```
>>> x = map(cube, range(1, 6))
```

Nothing is printed!
Iterators compute results
lazily

```
>>> x  
<map object at 0x10a002f20>
```

all and any

all(s: iterable) -> bool: Return True if all values in s evaluate to True
Returns as soon as a False value is reached

any(s: iterable) -> bool: Return True if at least one value in s evaluates to True
Returns as soon as a True value is reached

What's printed when evaluating:

```
>>> x = all(map(print, range(-3, 3)))
```

all and any

all(s: iterable) -> bool: Return True if all values in s evaluate to True
Returns as soon as a False value is reached

any(s: iterable) -> bool: Return True if at least one value in s evaluates to True
Returns as soon as a True value is reached

What's printed when evaluating:

```
>>> x = all(map(print, range(-3, 3)))
```

```
-3
```

Why?

- `print(-3)` returns `None` after displaying `-3`
- `None` is a false value
- `all([None, ...])` is `False` for any `...`
- The `map` iterator never needs to advance beyond `-3`

List Practice

Spring 2023 Midterm 2 Question 1

```
def chain(s):  
    return [s[0], s[1:]]  
silver = [2, chain([3, 4, 5])]  
gold = [silver[0], silver[1].pop()]  
silver[0] = 1  
platinum = chain(chain([6, 7, 8]))
```

Reminder: `s.pop()` removes and returns the last item in list `s`.

```
>>> silver
[1, [3]]
```

```
>>> gold
[2, [4, 5]]
```

```
>>> platinum
[6, [[7, 8]]]
```

