Iterators

Announcements

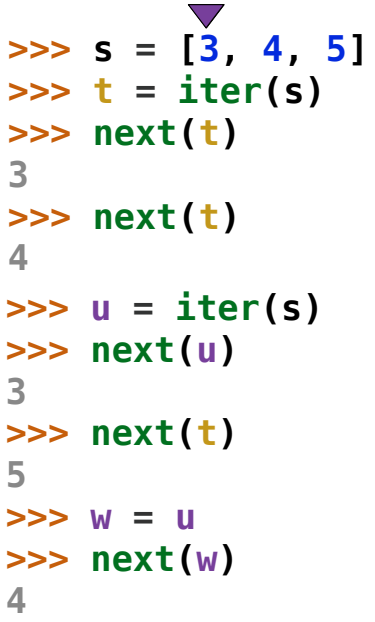
Iterators

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

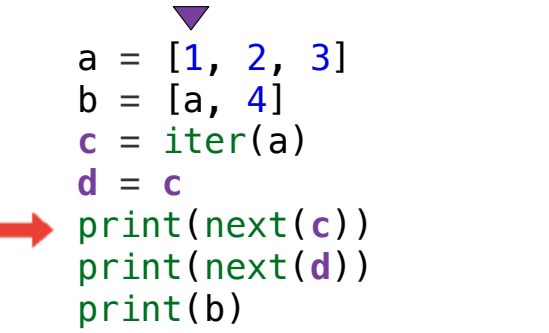
What will be printed?



```
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

```
                map(func, iterable):     Iterate over func(x) for x in iterable
             filter(func, iterable):     Iterate over x in iterable if func(x)
         zip(first_iter, second_iter):   Iterate over co-indexed (x, y) pairs
                reversed(sequence):      Iterate over x 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
    """
    total, k = 0, 1
    while k <= n:
        total = total + term(k)
        k = k + 1
    return total
```

0 + 1 + 8 + 27 + 64 + 125

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
    """
    total, k = 0, 1
    while k <= n:
        total = total + term(k)
        k = k + 1
    return total
```

0 + 1 + 8 + 27 + 64 + 125

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 / Any

all and any

all(s: iterable) -> bool: Return True if bool(x) is True for all values x in s

any(s: iterable) -> bool: Return True if bool(x) is True for at least one value x in s

```
>>> all([True, True, True])
True
```

All values are True

```
>>> all([[1], [1, 2], [1, 2, 3]])
True
```

All values are true values

```
>>> all([[], [1], [1, 2, 3]])
False
```

Empty list is a false value

```
>>> any([[], [1], [1, 2, 3]])
True
```

At least one value is a true value

```
def is_leafy(t: Tree) -> bool:
    """Return true if all of t's branches
       are leaves. """
    return all(map(_________ , _________________))
```

- Get the list of branches for the tree: `t.branches`
- Get the branch at index i: `t.branches[0]`
- Determine whether the tree is a leaf: `is_leaf(t)`

all and any

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

any(s: iterable) -> bool: Return True if bool(x) is True for at least one value x in s
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 bool(x) is True for all values x in s
Returns as soon as a false value is reached

any(s: iterable) -> bool: Return True if bool(x) is True for at least one value x in s
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 and stops checking
- The map iterator never needs to advances beyond -3