

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
>>> next(u)
4
```

(Demo)

Iterators vs. Iterables

Iterable – any object that can be iterated over i.e. sequences

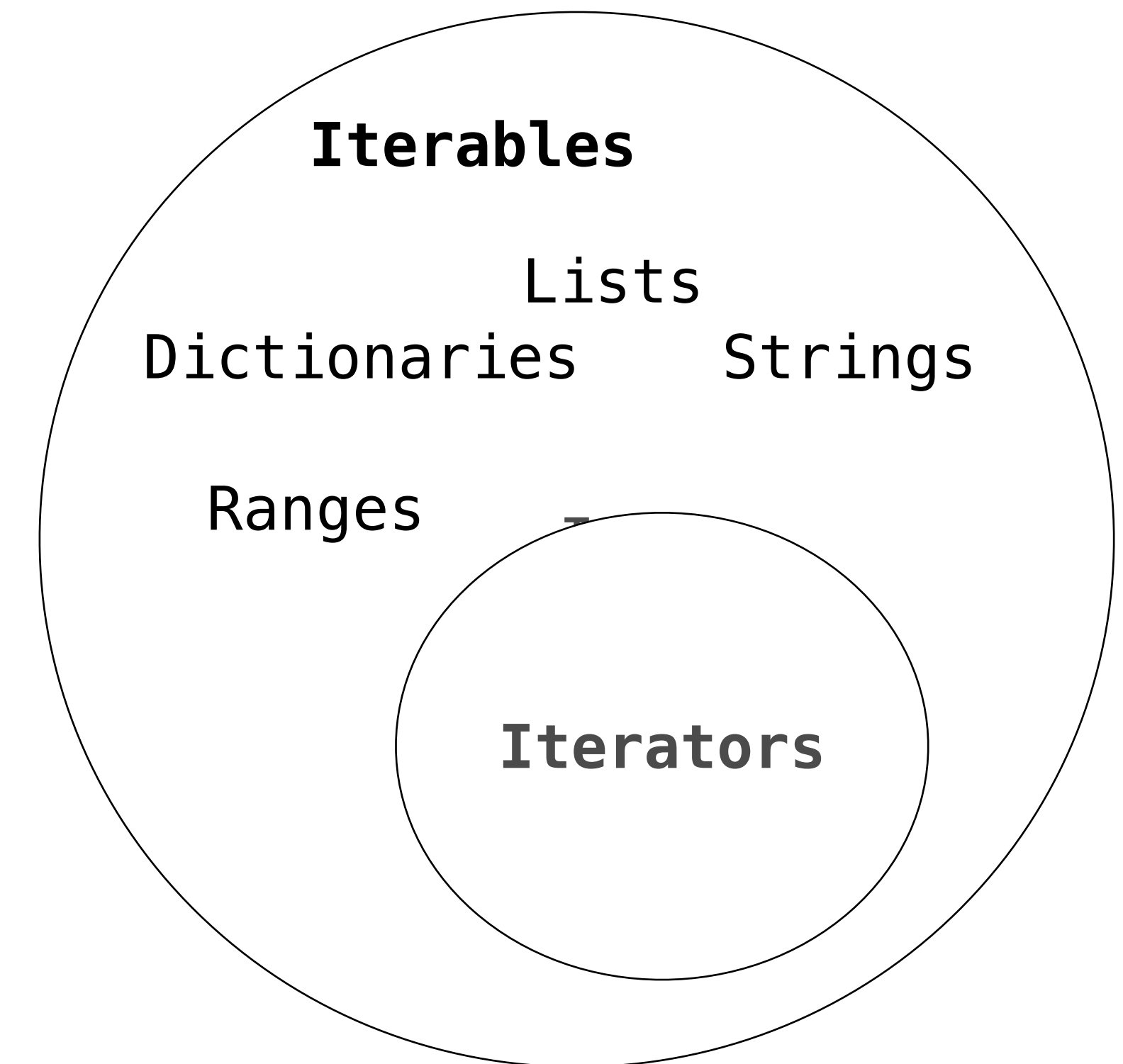
- Ex: Lists, Dictionaries, Strings, Ranges
- Usage: for loops, `iter(iterable)`

Iterator – can be created from iterables

- Usage: `next(iterator)`

All iterators are iterables. Not all iterables are iterators.

Calling `iter` on an iterator will return itself.



Dictionary Iteration

Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable

A dictionary, its keys, its values, and its items are all iterable values

- The order of items in a dictionary is the order in which they were added (Python 3.6+)
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```

For Statements

(Demo)

Python Implementation of For Loops

Pseudocode for for loops:

```
for x in <iterable>:  
    print(x)
```

```
it = iter(<iterable>)  
while there is no StopIteration:  
    x = next(it)  
    print(x)
```

Summary: For loops are just shorthand for creating an iterator and calling next on it until there are no more values left.

- We can for loop over an object if and only if it is an iterable (we must call `iter` on it)
- For loops will “use up” an iterator because it calls `next` on it until there are no more values

Built-In Iterator Functions

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

To view the contents of an iterator, place the resulting elements into a container

<code>list(iterable):</code>	Create a list containing all <code>x</code> in <code>iterable</code>
<code>tuple(iterable):</code>	Create a tuple containing all <code>x</code> in <code>iterable</code>
<code>sorted(iterable):</code>	Create a sorted list containing <code>x</code> in <code>iterable</code>

(Demo)

Generators

Generators and Generator Functions

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3  
>>> t  
<generator object plus_minus ...>
```

A *generator function* is a function that **yields** values instead of **returning** them

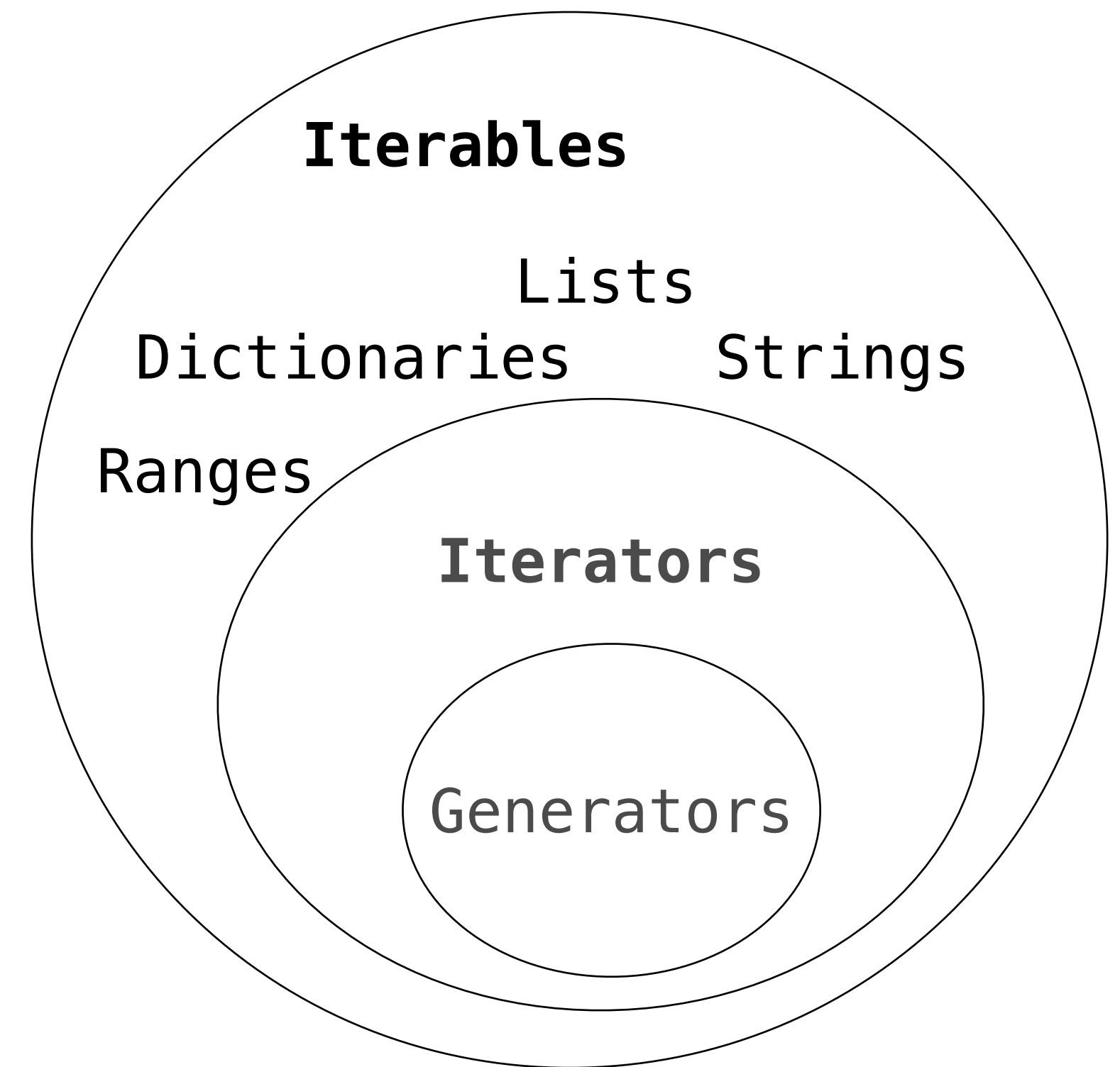
A normal function **returns** once; a *generator function* can **yield** multiple times

A *generator* is an iterator created automatically by calling a *generator function*

When a *generator function* is called, it returns a *generator* that iterates over its yields

(Demo)

Generators & Iterables



Generator Functions can Yield from Iterables

A **yield from** statement yields all values from an iterator or iterable (Python 3.3)

```
>>> list(a_then_b([3, 4], [5, 6]))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:  
        yield k  
        yield from countdown(k-1)
```

(Demo)

Infinite Generators

Generators can yield infinite sequences.

```
def gen_naturals():  
    i = 0  
    while True:  
        yield i  
        i += 1
```

```
>>> g = gen_naturals()  
>>> next(g)  
0  
>>> next(g)  
1
```

(Demo)