

# Iterate, Iterate, Iterate

Moshe Zadka – <https://cobordism.com>

# Iterable or Iterator

```
class Iterable(Protocol[T]):  
    def __iter__(self) -> Iterator[T]:  
        ...
```

# Iterable or Iterator

```
class Iterable(Protocol[T]):  
    def __iter__(self) -> Iterator[T]:  
        ...
```

```
class Iterator(Iterable[T]):  
    def __next__(self) -> T:  
        """Raises StopIteration"""
```

# Iterable or Iterator

```
class Iterable(Protocol[T]):  
    def __iter__(self) -> Iterator[T]:  
        ...
```

```
class Iterator(Iterable[T]):  
    def __next__(self) -> T:  
        """Raises StopIteration"""
```

The `__iter__` method returns `self`

# Functions

```
def iter(thing):  
    return thing.__iter__()
```

```
def next(thing):  
    return thing.__next__()
```

# Making it iterable

```
@dataclass(frozen=True)
class TwoInts:
    thing_1: int
    thing_2: int

    def __iter__(self) -> Iterator[T]:
        return iter([self.thing_1, self.thing_2])
```

# Making it iterable

```
dec = TwoInts(1, 10)
iter_dec = iter(dec)
print("thing_1", next(iter_dec))
print("thing_2", next(iter_dec))
```

```
thing_1 1
thing_2 10
```

# Moving Forward

```
class Twolntlter:
    def __init__(self) -> None:
        self.things = [1, 10]

    def __next__(self) -> int:
        try:
            return self.things.pop(0)
        except IndexError:
            raise StopIteration() from None
```



# Moving Forward

```
iter_dec = TwoIntIter()  
print(next(iter_dec))  
print(next(iter_dec))
```

1

10

## Moving Forward: Usage

```
iter_dec = iter([1, 10])  
print(next(iter_dec))  
print(next(iter_dec))  
try:  
    next(iter_dec)  
except Exception as exc:  
    print("Stopped", repr(exc))
```

1

10

Stopped StopIteration()

## for: Using iterators

```
class Twolntlter:
    def __init__(self) -> None:
        self.things = [1, 10]

    def __iter__(self) -> Iterator[int]:
        return self

    def __next__(self) -> int:
        try:
            ret_value = self.things.pop(0)
        except IndexError:
            print("__next__: _End_of_iteration")
            raise StopIteration() from None
        print("__next__: _Returning", ret_value)
        return ret_value
```

## for: Using iterators

```
iter_dec = TwoIntIter()  
print("for: _Before")  
for val in iter_dec:  
    print("for: _Got" , val)  
print("for: _After")
```

```
for: Before  
__next__: Returning 1  
for: Got 1  
__next__: Returning 10  
for: Got 10  
__next__: End of iteration  
for: After
```

## Sequence constructors: Using iterators

```
res = list(TwoIntIter())  
print("Result is", res)
```

```
--next--: Returning 1  
--next--: Returning 10  
--next--: End of iteration  
Result is [1, 10]
```

# Generator vs. Iterator

Iterator: Protocol

# Generator vs. Iterator

Iterator: Protocol

Annoying to implement (state)

# Generator vs. Iterator

Iterator: Protocol

Annoying to implement (state)

Generator: Mechanism to build iterators



# Generators and yield

```
def two_int_iter():  
    print(" Begin")  
    yield 1  
    print(" Middle")  
    yield 10  
    print(" End")
```

# Generators and yield

```
iter_dec = two_int_iter()  
print("for: _Before")  
for val in iter_dec:  
    print("for: _Got" , val)  
print("for: _After")
```

```
for: Before  
Begin  
for: Got 1  
Middle  
for: Got 10  
End  
for: After
```

## yield from

```
def three_int_iter():  
    yield from two_int_iter()  
    yield 100
```

```
list(three_int_iter())
```

Begin

Middle

End

```
[1, 10, 100]
```

# Mindless consumption

```
from collections import deque
```

```
deque(two_int_iter(), maxlen=0)
```

Begin

Middle

End

```
deque([], maxlen=0)
```

# Iterable to Iterator

# Iterable to Iterator

Input: any iterable

# Iterable to Iterator

Input: any iterable

Output: on-demand iterator

## enumerate

```
list (enumerate (iter ([1, 10, 100])))  
[(0, 1), (1, 10), (2, 100)]
```



## map

```
from operator import add  
from functools import partial
```

```
list(map(  
    partial(add, 1),  
    iter([10, 20]),  
))
```

```
[11, 21]
```

## filter

```
from operator import ge  
from functools import partial
```

```
list( filter(  
    partial(ge, 5),  
    iter([1, 10, 3]),  
) )
```

```
[1, 3]
```

## zip

```
list(zip(  
    iter([10, 100]),  
    iter([2, 4]),  
))
```

```
[(10, 2), (100, 4)]
```

## zip

```
double_plus_one = (  
    2 * a + 1  
    for a in  
        iter([1, 10])  
)  
  
list(double_plus_one)  
  
[3, 21]
```

# itertools

# itertools

standard

# itertools

standard

Advanced iterator algebra

## chain

```
list(  
    itertools.chain(  
        iter([1, 2, 3]),  
        iter([4, 5]),  
    )  
)  
  
[1, 2, 3, 4, 5]
```



## islice

```
list(itertools.islice(two_int_iter(), 0, 1))
```

Begin

```
[1]
```

count

```
list(itertools.islice(itertools.count(), 0, 5))
```

```
[0, 1, 2, 3, 4]
```

# more-itertools

# more-itertools

When itertools is not enough

## chunked

```
list(more_itertools.chunked(iter([0, 1, 2, 3]), 2))  
[[0, 1], [2, 3]]
```

## distribute

```
evens, odds = more_itertools.distribute(  
    2,  
    iter([0, 1, 2, 3]),  
)  
print(list(evens))  
print(list(odds))
```

[0, 2]

[1, 3]

## peekable

```
some_nums = more_itertools.peekable(  
    iter([0, 1, 2, 3, 4])  
)  
  
for x in some_nums:  
    if some_nums.peek() == 3:  
        break  
    print(x)
```

0

1

## windowed

```
samples = more_itertools.windowed(  
    iter([0, 3, 7, 10, 12]),  
    2  
)
```

```
for start, end in samples:  
    print(end - start)
```

3

4

3

2



## Iterators are (weirdly) mutable

```
things = iter([1, 2, 3])  
print("first_time", " ".join(map(str, things)))  
print("second_time", " ".join(map(str, things)))
```

```
first time 1 2 3  
second time
```

## Iterator mutation: solution one

```
things = [1, 2, 3]
print("first_time", " ".join(map(str, iter(things))
print("second_time", " ".join(map(str, iter(things))
```

```
first time 1 2 3
second time 1 2 3
```

## Iterator mutation: solution two

```
things = iter([1, 2, 3])
saved_things = list(things)
print("first_time", " ".join(map(str, saved_things)))
print("second_time", " ".join(map(str, saved_things)))
```

```
first time 1 2 3
second time 1 2 3
```

## Dictionary, iteration, and mutation

```
things = iter(stuff := dict(a=1, b=2))
stuff.pop("a")
try:
    list(things)
except Exception as exc:
    print(repr(exc))
```

RuntimeError('dictionary changed size during iteration')

## The last yield matters

```
def gen():  
    yield "hello"  
    print("going")  
    yield "goodbye"  
print(type(gen()))
```

```
<class 'generator'>
```

## The last yield matters

```
def gen():  
    # yield "hello"  
    print("going")  
    yield "goodbye"  
print(type(gen()))
```

```
<class 'generator'>
```

## The last yield matters

```
def gen():  
    # yield "hello"  
    print("going")  
    # yield "goodbye"  
print(type(gen()))
```

```
going  
<class 'NoneType'>
```

## The last yield matters

```
def gen():  
    if False:  
        yield # Fake  
        # yield "hello"  
        print("going")  
        # yield "goodbye"  
print(type(gen()))
```

```
<class 'generator'>
```



# Iterators and Generators

# Iterators and Generators

Iterators: Useful interface

# Iterators and Generators

Iterators: Useful interface

Generators: Useful way to implement interface

# Algebra of Just-in-time

# Algebra of Just-in-time

Functions that accept and return iterators

# Algebra of Just-in-time

Functions that accept and return iterators  
Composable just-in-time tools

# Ecosystem

# Ecosystem

Research



# Ecosystem

Research  
Consume

# Ecosystem

Research  
Consume  
Contribute