Iterators + Generators

Iterators

Iterables

Lists, tuples, dictionaries, and strings are all **iterable** objects.

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]

ranked_chocolates = ("Dark", "Milk", "White")

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}

best_topping = "pineapple"
```

Iterating

We can iterate over iterable objects:

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
for item in my_order:
    print(item)
lowered = [item.lower() for item in my_order]

ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
for product in prices:
    print(product, " costs ", prices[product])
discounted = { item: prices[item] * 0.75 for item in prices }

best_topping = "pineapple"
for letter in best_topping:
    print(letter)
```

Iterators

An **iterator** is an object that provides sequential access to values, one by one.

`iter(iterable)` returns an iterator over the elements of an iterable.

`next(iterator)` returns the next element in an iterator.

```
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]

topperator = iter(toppings)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
```

Iterators

An **iterator** is an object that provides sequential access to values, one by one.

`iter(iterable)` returns an iterator over the elements of an iterable.

`next(iterator)` returns the next element in an iterator.

```
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]

topperator = iter(toppings)
next(iter) # 'pineapple'
next(iter) # 'pepper'
next(iter) # 'mushroom'
next(iter) # 'roasted red pepper'
next(iter)
```

Iterators

An **iterator** is an object that provides sequential access to values, one by one.

`iter(iterable)` returns an iterator over the elements of an iterable.

`next(iterator)` returns the next element in an iterator.

```
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]

topperator = iter(toppings)
next(iter) # 'pineapple'
next(iter) # 'pepper'
next(iter) # 'mushroom'
next(iter) # 'roasted red pepper'
next(iter) # ❌ StopIteration exception
```

Handling StopIteration

An unhandled exception will immediately stop a program.

Use `try`/`except` to handle an exception:

```
ranked_chocolates = ("Dark", "Milk", "White")

chocolaterator = iter(ranked_chocolates)
print(next(chocolaterator))
print(next(chocolaterator))
print(next(chocolaterator))

try:
    print(next(chocolaterator))
except StopIteration:
    print("No more left!")
```

Iterating with Iterators

We can use a `while` loop to process iterators of arbitrary length:

```
ranked_chocolates = ("Dark", "Milk", "White")
chocolaterator = iter(ranked_chocolates)

try:
    while True:
        choco = next(chocolaterator)
        print(choco)
except StopIteration:
    print("No more left!")
```

Iterators vs. For loops

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator = iter(ranked_chocolates)

try:
    while True:
        choco = next(chocorator)
        print(choco)
except StopIteration:
    print("No more left!")
```

Why not just...

```
ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)
```

Iterators vs. For loops

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator = iter(ranked_chocolates)

try:
    while True:
        choco = next(chocorator)
        print(choco)
except StopIteration:
    print("No more left!")
```

Why not just...

```
ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)
```

Well, actually, a for loop is just syntactic sugar! 🍬

For loop execution

```
for <name> in <expression>:
    <suite>
```

1. Python evaluates `<expression>` to make sure it's an iterable.
2. Python gets an iterator for the iterable.
3. Python gets the next value from the iterator and assigns to `<name>`.
4. Python executes `<suite>`.
5. Python repeats until it sees a StopIteration error.

```
iterator = iter(<expression>)
try:
    while True:
        <name> = next(iterator)
        <suite>
except StopIteration:
    pass
```

Comparison

The sugary for loop: 🍬

```
ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)
```

The "look ma, no sugar" version: 🙌🏽

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator = ranked_chocolates.__iter__()
try:
    while True:
        print(chocorator.__next__())
except StopIteration:
    pass
```

Poll time! What do you prefer? ✍🏽

Behavior != Implementation

The for loop and iterator version behave the same, but the Python interpreter can choose to implement them in different ways, which can affect execution time.

Version	**10,000 runs**	**1,000,000 runs**
For loop	3.2 milliseconds	336 milliseconds
Iterator	8.3 milliseconds	798 milliseconds

Is that significant? 🤔

We typically use a `for` loop unless we have a particular reason to use `next()`/`iter()`/`StopIteration`, since it is both easier to read and better optimized.

Functions that return iterators

Function	Description
`reversed(sequence)`	Iterate over item in `sequence` in reverse order (See example in PythonTutor)
`zip(*iterables)`	Iterate over co-indexed tuples with elements from each of the `iterables` (See example in PythonTutor)
`map(func, iterable, ...)`	Iterate over `func(x)` for `x` in `iterable` (See example in PythonTutor)
`filter(func, iterable)`	Iterate over `x` in `iterable` if `func(x)` (See example in PythonTutor)

Functions that return iterators

Function	Description
`reversed(sequence)`	Iterate over item in `sequence` in reverse order (See example in PythonTutor)
`zip(*iterables)`	Iterate over co-indexed tuples with elements from each of the `iterables` (See example in PythonTutor)
`map(func, iterable, ...)`	Iterate over `func(x)` for `x` in `iterable` Same as `[func(x) for x in iterable]` (See example in PythonTutor)
`filter(func, iterable)`	Iterate over `x` in `iterable` if `func(x)` (See example in PythonTutor)

Functions that return iterators

Function	Description
`reversed(sequence)`	Iterate over item in `sequence` in reverse order (See example in PythonTutor)
`zip(*iterables)`	Iterate over co-indexed tuples with elements from each of the `iterables` (See example in PythonTutor)
`map(func, iterable, ...)`	Iterate over `func(x)` for `x` in `iterable` Same as `[func(x) for x in iterable]` (See example in PythonTutor)
`filter(func, iterable)`	Iterate over `x` in `iterable` if `func(x)` Same as `[x for x in iterable if func(x)]` (See example in PythonTutor)

A useful detail

Calling `iter()` on an iterator just returns the iterator:

```
numbers = ["一つ", "二つ", "三つ"]
num_iter = iter(numbers)
num_iter2 = iter(num_iter)

assert num_iter is num_iter2
```

That's why this works...

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in filter(lambda x: x % 2 == 0, nums):
    print(num)
```

Functions that return iterables

Function	Description
`list(iterable)`	Create a list containing all items in `iterable`
`tuple(iterable)`	Create a tuples containing all items in `iterable`
`sorted(iterable)`	Create a sorted list containing all items in `iterable`

Generators

Generators

A **generator** is a type of iterator that yields results from a generator function.

A **generator function** uses `yield` instead of `return`:

```
def evens():
    num = 0
    while num < 10:
        yield num
        num += 2
```

Just call the generator function to get back a generator:

```
evengen = evens()

next(evengen)
next(evengen)
next(evengen)
next(evengen)
next(evengen)
next(evengen)
```

Generators

A **generator** is a type of iterator that yields results from a generator function.

A **generator function** uses `yield` instead of `return`:

```
def evens():
    num = 0
    while num < 10:
        yield num
        num += 2
```

Just call the generator function to get back a generator:

```
evengen = evens()

next(evengen)  # 0
next(evengen)  # 2
next(evengen)  # 4
next(evengen)  # 6
next(evengen)  # 8
next(evengen)  # ❌ StopIteration exception
```

How generators work

```
def evens():
    num = 0
    while num < 2:
        yield num
        num += 2

gen = evens()

next(gen)
next(gen)
```

- When the function is called, Python immediately returns an iterator without entering the function.
- When `next()` is called on the iterator, it executes the body of the generator from the last stopping point up to the next `yield` statement.
- If it finds a `yield` statement, it pauses on the next statement and returns the value of the yielded expression.
- If it doesn't reach a yield statement, it stops at the end of the function and raises a `StopIteration` exception.

Looping over generators

We can use for loops on generators, since generators are just special types of iterators.

```
def evens(start, end):
    num = start + (start % 2)
    while num < end:
        yield num
        num += 2

for num in evens(12, 60):
   print(num)
```

Looping over generators

We can use for loops on generators, since generators are just special types of iterators.

```
def evens(start, end):
    num = start + (start % 2)
    while num < end:
        yield num
        num += 2

for num in evens(12, 60):
   print(num)
```

Looks a lot like...

```
evens = [num for num in range(12, 60) if num % 2 == 0]
# Or  = filter(lambda x: x % 2 == 0, range(12, 60))
for num in evens:
    print(num)
```

Why use generators?

Generators are lazy: they only generate the next item when needed.

Why generate the whole sequence...

```
def find_matches(filename, match):
    matched = []
    for line in open(filename):
        if line.find(match) > -1:
            matched.append(line)
    return matched

matched_lines = find_matches('frankenstein.txt', "!")
matched_lines[0]
matched_lines[1]
```

...if you only want some elements?

```
def find_matches(filename, match):
    for line in open(filename):
        if line.find(match) > -1:
            yield line

line_iter = find_matches('frankenstein.txt', "!")
next(line_iter)
next(line_iter)
```

A large list can cause your program to run out of memory!

Yielding from iterables

A `yield from` statement yields the values from an iterator one at a time. 🍬

Instead of...

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

list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

We can write...

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

list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

Recursive yield from

A `yield from` can also yield the results of a generator function.

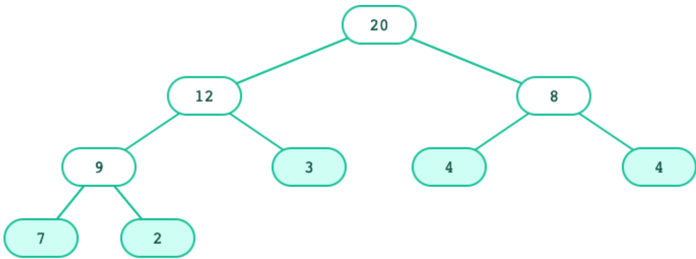
Instead of...

```
def factorial(n, accum):
    if n == 0:
        yield accum
    else:
        for result in factorial(n - 1, n * accum):
            yield result

for num in factorial(3, 1):
    print(num)
```

We can write...

```
def factorial(n, accum):
    if n == 0:
        yield accum
    else:
        yield from factorial(n - 1, n * accum)

for num in factorial(3, 1):
    print(num)
```

Recursive generators for trees



A pre-order traversal of the tree leaves:

```
def leaves(t):
    yield label(t)
    for c in branches(t):
        yield from leaves(c)

t = tree(20, [tree(12,
               [tree(9,
                  [tree(7), tree(2)]),
                tree(3)]),
              tree(8,
               [tree(4), tree(4)])])

leave_gen = leaves(t)
next(leave_gen)
```