Programmieren I (Python)

Christian Osendorfer

2023-12-07

Iterators

Iterable

- *Iterable* objects:
  - List, Tuple, Dictionary, String, `range`
- Iterables hold data that we want to *iterate* over one value at a time.
  - This is done using a `for` loop.
  - Pure iterables hold the data themselves.
- An iterable implements the **iterable protocol**:
  - The built-in function `iter` returns an **iterator**.
    - An iterable implements the `__iter__()` method.

Iterator

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.

```python
# What is happening with this snippet?
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]
topperator = iter(toppings)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
```

- `StopIteration`: **Signal** the end of the iterator!

`iter`

- Calling `iter` on an iterator just returns the iterator!
- `iter` returns an iterator for any iterable object!

```python
menue = ["Pie", "Pasta", "Pillow"]
iter_m = iter(menue)
next_m = next(iter_m)
what_is_this = next(iter(next_m))
```

- In a Dictionary, its keys, its values and its items are all iterable values!
  - Since Python 3.6, the order of items in dictionaries is the order in which they were added.

```python
d = {'one': 1, 'two': 2}
d['zero'] = 0
k, v, i = iter(d.keys()), iter(d.values()), iter(d.items())
```

For loop execution – revisited

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

1. Python evaluates to make sure it’s iterable.

2. Python gets an iterator for the iterable.

3. Python gets the next value from the iterator and assigns to .

4. Python executes .

5. Python repeats until it sees a `StopIteration` error.

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

For loop with iterator

- When used in a `for` loop, Python will call `next()` on the iterator in each iteration:

```python
nums = range(1, 4)
nums_iter = iter(nums)
for num in nums_iter:
  print(num)
```

- Iterators are mutable! Once an iterator moves forward, it won’t return the values that came before.

```python
# ... continue from above
for num in nums_iter:
  print(num)
for num in nums: # range is an iterable, not iterator!
  print(num)
```

Reasons for using iterators

- A code that processes an iterator using `iter()` or `next()` makes few assumptions about the data itself.
  - Changing the data storage from a *list* to a *tuple*, or a *dict* doesn't require rewriting code.
  - Others are more likely to be able to use your code on their data.
- An iterator **bundles together** a *sequence* and a *position within the sequence* in a **single object**.
  - Passing that object to another function always retains its position.
  - Ensures that each element of the sequence is only processed once.
  - Limits the operations that can be performed to only calling `next()`.
- An iterator **abstracts away** the underlying memory specification for a sequence.
  - It (will) allow(s) us to handle infinite sequence-like data structures (combined with **generators**).

Functions that return iterables

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

- `list(iterable)`: Create a list containing all x in `iterable`.
- `tuple(iterable)`: Create a tuple containing all x in `iterable`.
- `sorted(iterable)`: Create a sorted list containing all x in `iterable`.
  - Elements must be comparable!

Functions that return iterators

- `map(func, iterable, ...)`: Iterate over `func(x)` for `x` in `iterable`.
  - Same as `[func(x) for x in iterable]`. See PythonTutor.
- `filter(func, iterable)`: Iterate over `x` in `iterable` if `func(x)` is `True`.
  - Same as `[x for x in iterable if func(x)]`. `func` is called a *predicate*.
- `zip(*iterables)`: Iterate over co-indexed tuples with elements from each of the `iterables`.
  - See PythonTutor.
  - Investigate the notation `*iterables` as the parameter(s) to `zip`.
- `reversed(sequence)`: Iterate over item in `sequence` in reverse order.
  - `sequence` must be a finite iterator!

Built-in `map` function

- `map(func, iterable)`: Applies func(x) for x in iterable and returns an iterator

```python
def double(num):
    return num * 2

for num in map(double, [1, 2, 3]):
    print(num)

for word in map(lambda text: text.lower(), ["SuP", "HELLO", "Hi"]):
    print(word)
```

Built-in `filter` function

- `filter(func, iterable)`: Returns an iterator from the items of iterable where func(item) is true.

```python
def is_fourletterword(text):
    return len(text) == 4

for word in filter(is_fourletterword, ["braid", "bode", "brand", "band"]):
    print(word)

for num in filter(lambda x: x % 2 == 0, [1, 2, 3, 4]):
    print(num)
```

Built-in zip function

- `zip(*iterables)`: Returns an iterator that aggregates elements from each of the iterables into co-indexed pairs

```python
# ["one", "two", "three"]   --> ("one", "uno")  ("two", "dos")  ("three", "
# ["uno", "dos", "tres"]

english_nums = ["one", "two", "three"]
spanish_nums = ["uno", "dos", "tres"]
german_nums = ["eins", "zwei"]
zip_iter = zip(english_nums, spanish_nums, german_nums)
english, spanish, german = next(zip_iter)
print(english, spanish, german)

for english, spanish, german in zip(english_nums, spanish_nums, german_nums
    print(english, spanish, german)
```

Generators

Generators

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

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

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

Just **call the generator function** to **get back a generator**:

```python
evengen = evens()

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

How generators work

```python
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.

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

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

Why use generators?

- Generators are **lazy**: they only generate the next item when needed.
- Why generate the whole sequence…

```python
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?

```python
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!

Examples: Countdown

```python
def countdown(n):
    """
    Generate a countdown of numbers from N down to 'blast off!'.
    >>> c = countdown(3)
    >>> next(c)
    3
    >>> next(c)
    2
    >>> next(c)
    1
    >>> next(c)
    'blast off!'
    """
    while n > 0:
        yield n
        n -= 1
    yield "blast off!"
```

Examples: Virahanka-Fibonacci generator

Let’s transform this function…

```python
def virfib(n):
    """Compute the nth Virahanka-Fibonacci number, for N >= 1.
    >>> virfib(6)
    8
    """
    prev = 0  # First Fibonacci number
    curr = 1  # Second Fibonacci number
    k = 1
    while k < n:
        (prev, curr) = (curr, prev + curr)
        k += 1
    return curr
```

..into a generator function!

Examples: Virahanka-Fibonacci generator

```python
def generate_virfib():
    """Generate the next Virahanka-Fibonacci number.
    >>> g = generate_virfib()
    >>> next(g)
    0
    >>> next(g)
    1
    >>> next(g)
    1
    >>> next(g)
    2
    """
    prev = 0  # First Fibonacci number
    curr = 1  # Second Fibonacci number
    while True:
        yield prev
        (prev, curr) = (curr, prev + curr)
```

`Yield from` iterables

- A `yield from` **statement** can be used to yield the values from an iterable one at a time.
- Instead of…

```python
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…

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

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

Yielding from generators

A `yield from` can also yield the results of another generator function (which could **be itself**).

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

Generator function with a `return`

When a generator function executes a `return` statement, it exits and cannot yield more values.

```python
def f(x):
    yield x
    yield x + 1
    return
    yield x + 3
```

https://haw-landshut.de/ki