Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

RLE
Generator Comprehension
Itertools

AGENDA

- RLE
- Generator Comprehension
- Itertools

RLE

- Definition
- Example

DEFINITION

- Run-length encoding
- Form of lossless data compression
  - Allows the original data to be perfectly reconstructed from the compressed data with no loss of information
- Sequences in which the same data value occurs in many consecutive data elements are stored as a single data value and count
- Simple graphic images such as icons, line drawings, animations, …
- For files without many runs, RLE could increase the size

EXAMPLE

wwwwaaadexxxxxx

w4a3d1e1x6

EXERCISE

- Try the following exercises

- 08-rle

Generator Comprehension

- Definition
- Example
- Coded similar to list comprehensions
- Generate list using generator expressions

DEFINITION

- Generators are written just like a normal function but we use yield instead of return for returning a result
- Evaluation of elements on demand
- When the function terminates, StopIteration is raised automatically on further calls
- Takes much less memory than a list

EXAMPLE

```
def generator():
    t = 1
    yield t

    t += 1
    yield t

    t += 1
    yield t

call = generator()
print(next(call))
# result: 1
print(next(call))
# result: 2
print(next(call))
# result: 3
next(call)
# Error StopIteration
```

CODED SIMILAR TO LIST COMPREHENSIONS

```
generator = (num ** 2 for num in range(10))
for num in generator:
    print(num)
'''
Result:
0
1
4
9
16
25
...
'''
```

```
generator = (num ** 2 for num in range(10))
print(next(generator))
# result: 0
print(next(generator))
# result: 1
print(next(generator))
# result: 4
print(next(generator))
# result: 9
print(next(generator))
# result: 16
...
```

- Coded like list comprehension but instead of brackets, use parenthesis
- Designed for situations where the generator is used right away by an enclosing function

GENERATE LIST USING GENERATOR EXPRESSIONS

```
string = 'Programming 2'
li = list(string[i] for i in range(len(string)-1, -1, -1))
print(li)
# result: ['2', ' ', 'g', 'n', 'i', 'm', 'm', 'a', 'r', 'g', 'o', 'r', 'P']
```

EXERCISE

- Try the following exercises

- 09-generator-comprehension

Itertools

- Definition
- Example
- Different types of iterators
- References

DEFINITION

- Itertool is a module that provides various functions that work on iterators to produce complex iterators
- Fast, memory-efficient tool

EXAMPLE

```
import operator
import time

L1 = [1, 2, 3, 4, 5, 6]
L2 = [2, 3, 4, 5, 6, 7]

t1 = time.perf_counter()
for i in range(6):
    print(L1[i] * L2[i], end=" ")
t2 = time.perf_counter()

print("\nTime taken by for loop: %.8f" % (t2 - t1))
# Result:
# 2 6 12 20 30 42
# Time taken by for loop: 0.00001980
```

```
import operator
import time

L1 = [1, 2, 3, 4, 5, 6]
L2 = [2, 3, 4, 5, 6, 7]

t1 = time.perf_counter()
a, b, c, d, e, f = map(operator.mul, L1, L2)
t2 = time.perf_counter()

print(a, b, c, d, e, f)
print("Time taken by map function: %.8f" % (t2 - t1))
# Result:
# 2 6 12 20 30 42
# Time taken by map function: 0.00000410
```

- Iterating through the elements on both the list simultaneously and multiplying them is a naïve approach
- Better is to use the map function by passing the mul operator and both lists

DIFFERENT TYPES OF ITERATORS

- Infinite iterators
- Combinatoric iterators
- Terminating iterators

DIFFERENT TYPES OF ITERATORS: INFINITE

- Not necessary that an iterator object has to exhaust, sometimes it can be infinite
- There are three types of infinite iterators:
  - count(start, step)
  - cycle(iterable)
  - repeat(val, num)

DIFFERENT TYPES OF ITERATORS: INFINITE: COUNT EXAMPLE

```
import itertools

for i in itertools.count(5, 5):
    if i == 35:
        break
    else:
        print(i, end=" ")
# result: 5 10 15 20 25 30
```

- Starts printing from the start parameter number and prints infinitely
- Possible to add steps

DIFFERENT TYPES OF ITERATORS: COMBINATORIC

- The recursive generators that are used to simplify combinatorial constructs such as permutations, combinations, and Cartesian products
- There are four combinatoric iterators:
  - Product()
  - Permutations()
  - Combinations()
  - Combinations_with_replacement()

DIFFERENT TYPES OF ITERATORS: COMBINATORIC: PRODUCT EXAMPLE

```
from itertools import product

print(list(product([1, 2], repeat=2)))
# Result: [(1, 1), (1, 2), (2, 1), (2, 2)]

print(list(product(['geeks', 'for', 'geeks'], '2')))
# Result: [('geeks', '2'), ('for', '2'), ('geeks', '2')]

print(list(product('AB', [3, 4])))
# Result: [('A', 3), ('A', 4), ('B', 3), ('B', 4)]
```

- Computes the cartesian product of input iterables
- Optional repeat keyword argument can be used to compute the product with itself

DIFFERENT TYPES OF ITERATORS: TERMINATING

- These iterators are used to work on the short input sequences and produce the output based on the functionality of the method used.
- Different types of terminating iterators are:
  - accumulate(iter, func)
  - chain(iter1, iter2…)
  - chain.from_iterable()
  - compress(iter, selector)
  - dropwhile(func, seq)
  - …

DIFFERENT TYPES OF ITERATORS: TERMINATING: ACCUMULATE EXAMPLE

```
import itertools
import operator

li1 = [1, 4, 5, 7]

print(list(itertools.accumulate(li1)))
# result: [1, 5, 10, 17]

print(list(itertools.accumulate(li1, operator.mul)))
# result: [1, 4, 20, 140]
```

- Takes two arguments, iterable target and the function which would be followed at each iteration of value in target
- Addition takes place by default

REFERENCES

- Python Docs
- Examples of other itertools

EXERCISE

- Try the following exercises

- 10-itertools