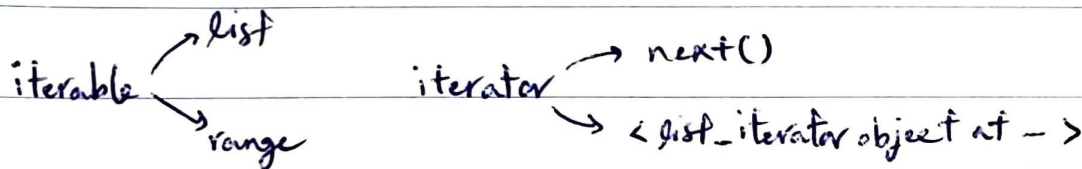


Python Data Science Toolbox (Part 2)

Iterators in Python Land:



iterator = iter(iterable) → Flash = ["Jack", "Barry", ...] → superspeed = iter(Flash)
not all iterables are actual lists → range(1) → creates a "range" object
print(next(superspeed))

iterable = range(3) → type: range
iterator = iter(range(3)) → type: range_iterator

e = enumerate(iterable) → items and index of each of them
e_list = list(e)
print(e_list) → [(0, "hi"), (1, "two"), (3, 39), ...] → for index, value in enumerate(e):
print(index, value)

for index, value in enumerate(e, start=12):

print(index, value) → 12, "hi"
13, "two"
⋮

enumerate() → list of tuples
zip() →

avengers = ['hawkeye', 'thor', ...]

names = ['barton', 'odinson', ...]

z = zip(avengers, names) → <class 'zip'>

z_list = list(z) → [('hawkeye', 'barton'), ...]

→ split operator

print(*z) → ('hawkeye', 'barton') () () ... → res1, res2 = zip(*z)

for z1, z2 in zip(avengers, names):
print(z1, z2)
hawkeye, barton
⋮

★ Loading data in chunks → too much data to hold in memory
→ solution: load data in chunks! → chunksize

```
import pandas as pd
```

```
result = []
```

iterable

each chunk is a DataFrame

```
for chunk in pd.read_csv('data.csv', chunksize = 1000):
```

```
    result.append(sum(chunk['x']))
```

```
total = sum(result)
```

چون جمع تپا بسیار زیاد است نمیتوانیم آن را در

لیست ذخیره کنیم؛ باید به صورت chunk ها مقادیر روی آن iterate کنیم!

List Comprehensions:

for loops are inefficient → coding time and space → one line of code

```
nums = [12, 8, 21, 3, 16]
```

iterable variable

```
new_nums = [num + 1 for num in nums]
```

output expression

iterable

with range()

we call it
predicate expression

```
result = [num for num in range(11)]
```

★ nested loops: `pairs = [(num1, num2) for num1 in range(0, 2) for num2 in range(6, 8)]`

Conditionals on the iterable: `[num * 2 for num in range(10) if num % 2 == 0]`

condition

✓ on the output expression: `[num * 2 if num % 2 == 0 else 0 for num in range(10)]`

dict Comprehensions → `pos_neg = {num: -num for num in range(9)}`

`{0: 0, 1: -1, 2: -2, ..., 8: -8}`

Intro to Generators:

```
(2 * num for num in range(10))
```

> <generator object>

list → []

→ returns list

Generator → ()

→ returns generator obj.

★ difference? Generator does not store the list in memory!

↳ great when we have very large sequence of data

```
result = (num for num in range(6))
```

```
print(next(result)) → 0
```

```
print(next(result)) → 1
```

⋮

→ 2

⋮

lazy evaluation: the evaluation of the expression is delayed until its value is needed

All the expressions used on iterators, such as `filter()`, can be used for generators!

★ Generator Function → like other functions, but we use "yield" instead of "return" to return a sequence of numbers ~~instead~~ as the output

```
def num_sequence(n):
```

```
    i = 0
```

```
    while i < n:
```

```
        yield i
```

```
        i += 1
```

```
result = num_sequence(5)
```

```
print(type(result)) → 'generator'
```

```
print(list(result)) → [0, 1, 2, 3, 4]
```

→ if we use "return", it would break out of function in the first iteration!

Generators are good for working on streaming data

when you open a connection to a file, the resulting file object is a generator! You can perform `file.readline()` like `generator.next()`.

→ using chunkSize to load large data in chunks:

```
[ df_reader = pd.read_csv(filename, chunksize=100)
  print(next(df_reader)) ]
```