ANALYSING THE LOOPING MECHANISMS

IN PYTHON LISTS

PyCascades '23

OUR PAIR

We are two software engineers passionate about exploring the hidden gems in Python.

agoda

arbisoft

SYED ANSAB WAQAR GILLANI

Software Engineer

SYED MUHAMMAD DAWOUD SHERAZ ALI

Senior Software Engineer

OUR AGENDA

Our content today is divided into three parts. Each part will be described with examples.

01 Python Loops

What loops are available in Python, and an example of how each loop is orchestrated.

02 Analysis

How different loops perform on similar kinds of operations and what complexities do they carry

03 Usage

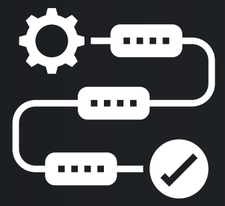
What loop to use and when?

ITERABLES IN PYTHON

An iterable is any object that can return its members one at a time

By definition, iterables support the iterator protocol, which specifies how object members are returned when an object is used in an iterator.

Python has two types of objects - iterator and iterables

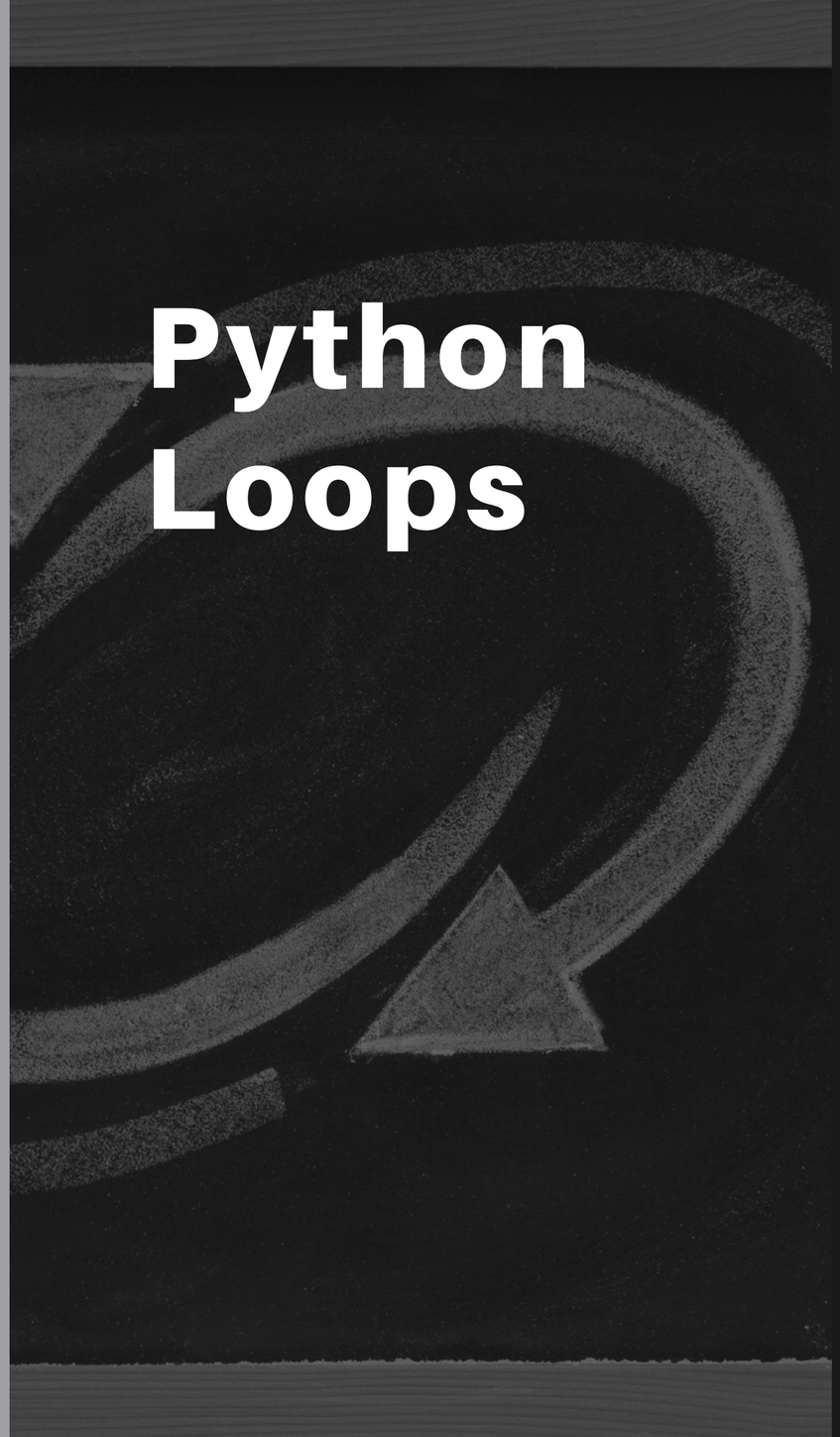


Sequence

An iterable which supports efficient element access using integer indices

Generator

A function that returns a generator iterator. It looks like a regular function except that it contains yield expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function.



Python Loops

FOR-IN ITERABLES LOOPS
for element in __iterable__:

RANGED LOOPS
for i in range(some_length):

ENUMERATE METHODS
for i, elem in enumerate(__iterable__):

WHILE LOOPS
while some_condition:

LIST COMPREHENSION
[i for i in __iterable__]

ZIP METHOD
for i in zip(iterable1, iterable2):

For-In iterables Loops

This type of loop iterates over a collection of objects rather than specifying numeric values or conditions:

```
def test_func():

    # creating mock iterable
    li = [1, 2, 3, 4, 5]

    # loop traversing for each element
    # in iterable
    for elem in li:
        # loop body
        elem += 1
```

```
5           0 BUILD_LIST               0
            2 LOAD_CONST               1 ((1, 2, 3, 4, 5))
            4 LIST_EXTEND              1
            6 STORE_FAST               0 (li)

6           8 LOAD_FAST                0 (li)
           10 GET_ITER
      >>   12 FOR_ITER                12 (to 26)
           14 STORE_FAST               1 (elem)

7          16 LOAD_FAST                1 (elem)
           18 LOAD_CONST               2 (1)
           20 INPLACE_ADD
           22 STORE_FAST               1 (elem)
           24 JUMP_ABSOLUTE           12
      >>   26 LOAD_CONST               0 (None)
           28 RETURN_VALUE
```

Ranged Loops

range() returns an object of class range, not a list or tuple of the values. Because a range object is an iterable, you can obtain the values by iterating over them with a for loop

```
def test_func():

    # loop traversing for inside a range of
    # 10 iterations, indexed 0 - 9
    for i in range(10):
        # loop body
        i += 1
```

```
 8           0 LOAD_GLOBAL              0 (range)
             2 LOAD_CONST               1 (10)
             4 CALL_FUNCTION            1
             6 GET_ITER
       >>    8 FOR_ITER                12 (to 22)
            10 STORE_FAST               0 (i)

10          12 LOAD_FAST                0 (i)
            14 LOAD_CONST               2 (1)
            16 INPLACE_ADD
            18 STORE_FAST               0 (i)
            20 JUMP_ABSOLUTE            8
       >>   22 LOAD_CONST               0 (None)
            24 RETURN_VALUE
```

```
 5           0 BUILD_LIST               0
             2 LOAD_CONST               1 ((1, 2, 3, 4, 5))
             4 LIST_EXTEND              1
             6 STORE_FAST               0 (li)

 8           8 LOAD_GLOBAL              0 (enumerate)
            10 LOAD_FAST                0 (li)
            12 CALL_FUNCTION            1
            14 GET_ITER
        >>  16 FOR_ITER                20 (to 38)
            18 UNPACK_SEQUENCE          2
            20 STORE_FAST               1 (count)
            22 STORE_FAST               2 (value)

10          24 LOAD_FAST                2 (value)
            26 LOAD_CONST               2 (1)
            28 BINARY_ADD
            30 LOAD_FAST                0 (li)
            32 LOAD_FAST                1 (count)
            34 STORE_SUBSCR
            36 JUMP_ABSOLUTE           16
        >>  38 LOAD_CONST               0 (None)
            40 RETURN_VALUE
```

Enumerate Methods

When you use enumerate(), the function gives you back two loop variables:

- The count of the current iteration
- The value of the item at the current iteration

```python
def test_func():
    li = [1, 2, 3, 4, 5]

    # loop enumerating each element
    # in list
    for count, value in enumerate(li):
        # loop body
        li[count] = value + 1
```

List comprehension

List comprehension is a Pythonic way to generate new lists from existing iterables. Sometimes, this technique is used for looping functions such as mapping and filtering

```
def test_func():
    # new_list = [
    #   expression for member in iterable
    #   if conditional
    # ]

    sentence = 'the rocket came back from mars'
    vowels = [i for i in sentence if i in 'aeiou']
    # ['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

```
          0 BUILD_LIST               0
          2 LOAD_FAST                0 (.0)
    >>    4 FOR_ITER                16 (to 22)
          6 STORE_FAST               1 (i)
          8 LOAD_FAST                1 (i)
         10 LOAD_CONST               0 ('aeiou')
         12 CONTAINS_OP              0
         14 POP_JUMP_IF_FALSE        4
         16 LOAD_FAST                1 (i)
         18 LIST_APPEND              2
         20 JUMP_ABSOLUTE            4
    >>   22 RETURN_VALUE
```

```
 6           0 BUILD_LIST               0
             2 LOAD_CONST               1 ((1, 2, 3, 4, 5))
             4 LIST_EXTEND              1
             6 STORE_FAST               0 (li)

 9           8 LOAD_CONST               2 (0)
            10 STORE_FAST               1 (i)

11     >>   12 LOAD_FAST                1 (i)
            14 LOAD_GLOBAL              0 (len)
            16 LOAD_FAST                0 (li)
            18 CALL_FUNCTION            1
            20 COMPARE_OP               0 (<)
            22 POP_JUMP_IF_FALSE       50

12          24 LOAD_FAST                0 (li)
            26 LOAD_FAST                1 (i)
            28 DUP_TOP_TWO
            30 BINARY_SUBSCR
            32 LOAD_CONST               3 (1)
            34 INPLACE_ADD
            36 ROT_THREE
            38 STORE_SUBSCR

13          40 LOAD_FAST                1 (i)
            42 LOAD_CONST               3 (1)
            44 INPLACE_ADD
            46 STORE_FAST               1 (i)
            48 JUMP_ABSOLUTE           12
       >>   50 LOAD_CONST               0 (None)
            52 RETURN_VALUE
```

While loop

When a while loop is encountered, <expr> is first evaluated in Boolean context. If it is true, the loop body is executed. Then <expr> is checked again, and if still true, the body is executed again. This continues until <expr> becomes false, at which point program execution proceeds to the first statement beyond the loop body.

```python
def test_func():
    # creating mock iterable
    li = [1, 2, 3, 4, 5]

    # creating variable for conditional
    i = 0

    while i < len(li):
        li[i] += 1
        i += 1
```

Zip Method

Python’s zip() function creates an iterator that will aggregate elements from two or more tables. You can use the resulting iterator to quickly and consistently solve common programming problems, like creating dictionaries.

```
def test_func():
    # creating mock iterable
    li1 = [1, 2, 3, 4, 5]
    li2 = [6, 7, 8, 9, 10]

    for elem1, elem2 in zip(li1, li2):
        elem1 += 1
        elem2 += 1
```

```
 6           0 BUILD_LIST               0
             2 LOAD_CONST               1 ((1, 2, 3, 4, 5))
             4 LIST_EXTEND              1
             6 STORE_FAST               0 (li1)

 7           8 BUILD_LIST               0
            10 LOAD_CONST               2 ((6, 7, 8, 9, 10))
            12 LIST_EXTEND              1
            14 STORE_FAST               1 (li2)

 9          16 LOAD_GLOBAL              0 (zip)
            18 LOAD_FAST                0 (li1)
            20 LOAD_FAST                1 (li2)
            22 CALL_FUNCTION            2
            24 GET_ITER
      >>    26 FOR_ITER                24 (to 52)
            28 UNPACK_SEQUENCE          2
            30 STORE_FAST               2 (elem1)
            32 STORE_FAST               3 (elem2)

10          34 LOAD_FAST                2 (elem1)
            36 LOAD_CONST               3 (1)
            38 INPLACE_ADD
            40 STORE_FAST               2 (elem1)

11          42 LOAD_FAST                3 (elem2)
            44 LOAD_CONST               3 (1)
            46 INPLACE_ADD
            48 STORE_FAST               3 (elem2)
            50 JUMP_ABSOLUTE           26
      >>    52 LOAD_CONST               0 (None)
            54 RETURN_VALUE
```

Analyzing the runtime on different loops

Sample Loop

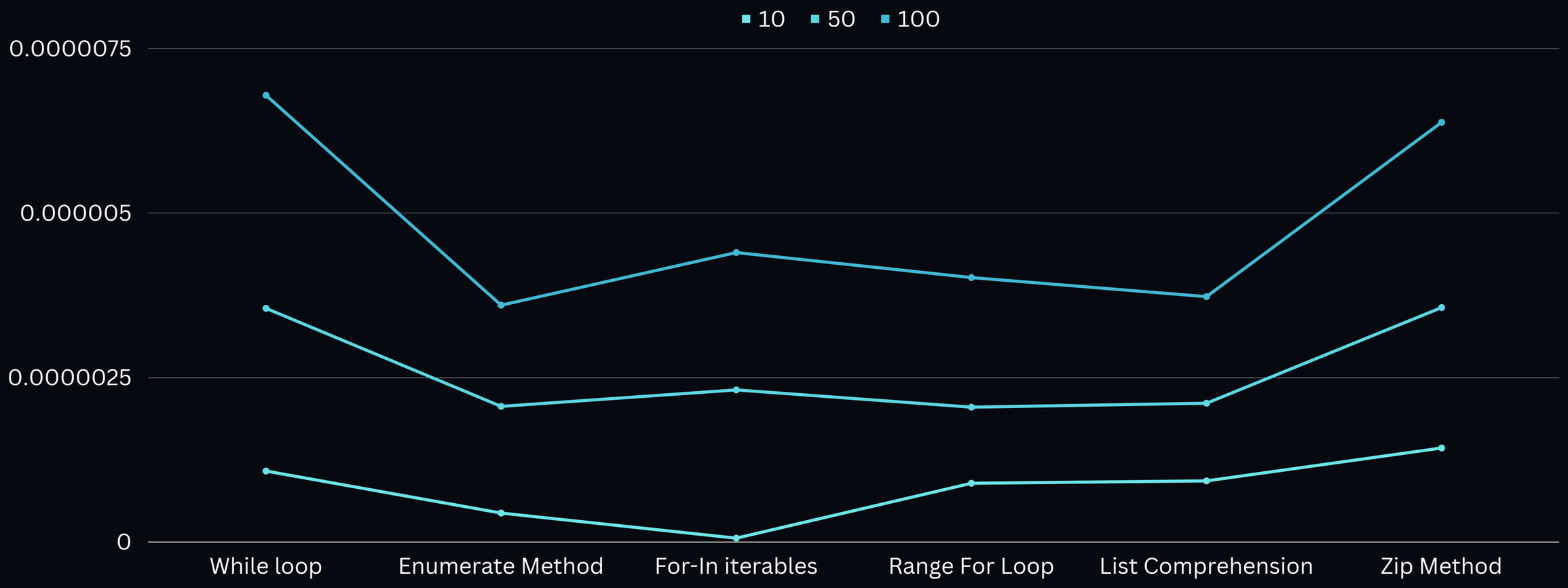
```python
def search_key(nums: list[int], key: int) -> list[int]:
    """
    Searching indexes for a key in a given list
    :param nums: list of integers
    :param key: integer to search for
    :return: all indexes of the key
    """
    indexes = []

    for i in range(len(nums)):
        if nums[i] == key:
            indexes.append(i)
    return indexes
```

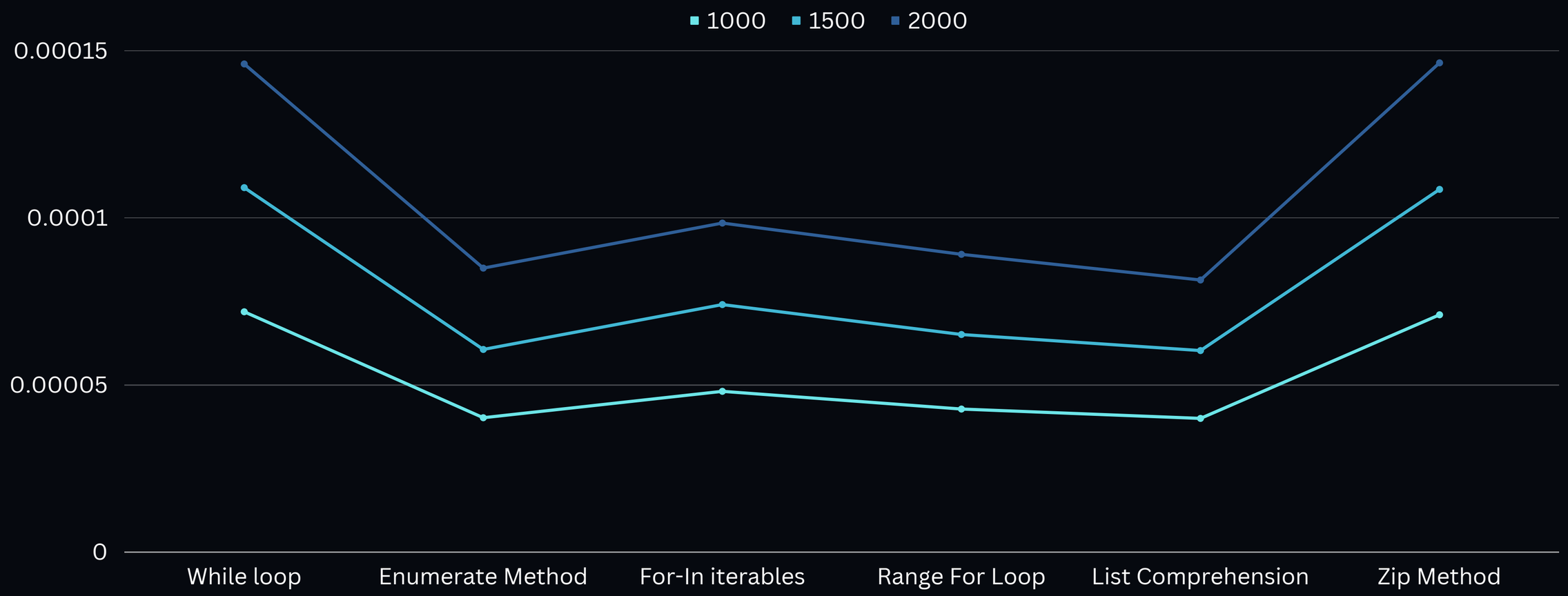
NOTE:

- All loops are executed with similar logic, minimizing any additional logic.
- All inputs are randomly generated, so there is no best or worst-case ratio.
- Every input is run on all loops, so there is no bias.
- All loop timings are average runtimes for each loop with 100 iterations.

Sample Size: Hundreds

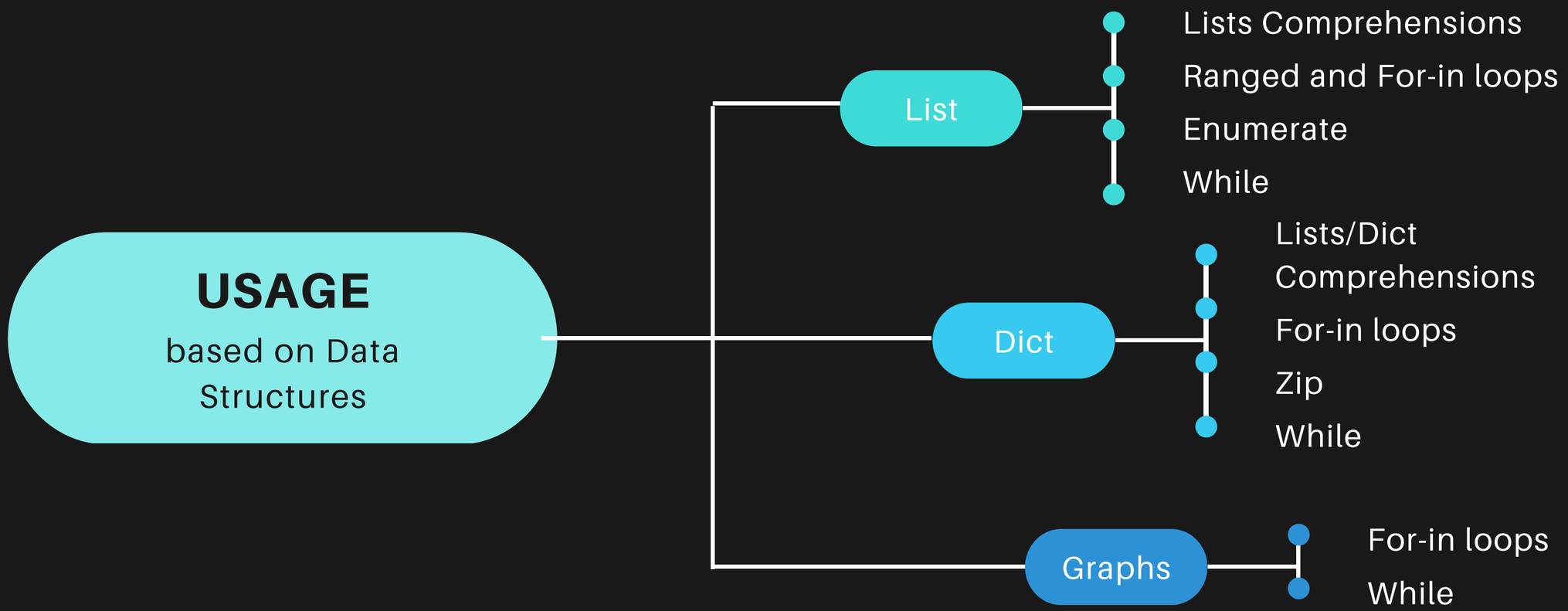


Sample Size: Thousands



Sample Size: Millions

What loops to use and when?


USAGE
based on Data Structures
List
Lists Comprehensions
Ranged and For-in loops
Enumerate
While
Dict
Lists/Dict Comprehensions
For-in loops
Zip
While
Graphs
For-in loops
While

USE CASES

How loops are used specific to particular use-cases

Although most loops are usually tied to specific data structures, there are some specific use cases that each loop is famous for

MULTIPLE LOOPING CONDITIONS

1

While loops are famous for being the only looping mechanism supporting multiple conditionals inside the loop.

CREATION

2

Although this operation can be done with any kind of loop, list comprehension has made a reputation for its extraordinary readability and optimization capabilities

MULTIPLE ITERABLES

3

Zip methods have established its rapport for connecting multiple iterables without breaking the bank of nested loops.

DATA - INDEX ASSOCIATION

4

enumerate for loops allow indexing the list contents without the need for managing a dedicated index, providing the capability to get any nth-index element from list on the fly

USE CASES

How loops are used specific to particular use-cases

Although most loops are
usually tied to specific data structures,
there are some specific use cases
that each loop is famous for

DATA PARSING

5

FOR-IN and Ranged loops offer a way to parse the list content without creating a new list

LAZY CONTENT PARSING

6

Generators have become a staple in reading the large content one at a time or in lazy fashion without overwhelming the memory, particularly helpful in ORMs like Django

References

- https://hsf-training.github.io/analysis-essentials/python/lists.html
- https://learnpython.com/blog/python-list-loop/
- https://www.geeksforgeeks.org/iterate-over-a-list-in-python/
- https://www.dataquest.io/blog/python-for-loop-tutorial/
- https://favtutor.com/blogs/python-iterate-through-list
- https://docs.python.org/3/library/dis.html
- https://deepsource.io/blog/python-performance-three-easy-tips/

THANK YOU

NOW WE'LL LEAVE ROOM FOR
Q&A