COMP1531

🔨10.1 - Python - Iterators & Generators

In this lecture

Why?

- Understand the concepts of iterators and iterables
- Create iterator classes
- Write simple generator functions
- Understand iterator invalidation
- Understand how python abstractions are implemented at a (slightly) lower level

How does a for loop actually work?

```python
shopping_list = ['apple', 'banana', 'pineapple', 'orange']

for item in shopping_list:
    print(item)
```

First attempt: C-style

```python
shopping_list = ['apple', 'banana', 'pineapple', 'orange']

for i in range(len(shopping_list)):
    print(shopping_list[i])
```

What if we don't know the length?

```python
from itertools import cycle

my_cycle = cycle([1, 2, 3])

for i in my_cycle:
    print(i)
```

Iterators

- An **iterator** is an object that enables a programmer to traverse a container
- Allows us to access the contents of a data structure while abstracting away its underlying representation
- In python, for loops are an abstraction of iterators
- Iterators can tell us:
  - Do we have any elements left?
  - What is the next element?

Let's rewrite our for-loop using an iterator

Iterators vs Iterables

- An **iterable** is an object that can be iterated over
- All iterators are iterable, but not all iterables are iterators
- For loops only need to be given something *iterable*
- Concretely:
  - An iterator has an __iter__() and __next__() methods
  - An iterable has an __iter__() method
- The __iter__() method
  - Returns an object of type iterator
- The __next__() method
  - Returns the next element in iteration
  - Raises a StopIteration if there are no elements left

A Custom Iterator: Square Numbers

Generators

- A functional way of writing iterators
- Defined via generator functions instead of classes
- Example generator

```python
def shopping_list():
    print(1)
    yield 'apple'
    print(2)
    yield 'orange'
    print(3)
    yield 'banana'
    print(4)
    yield 'pineapple'

for item in shopping_list():
    print(item)
```

Generators

- Intuitively, you can think of a generator as a suspendable computation
- Calling next() on a generator executes it until it reaches a yield, at which point it is suspended (frozen) until the subsequent call to next()

Generators

- More useful examples

```python
def squares():
    i = 0
    while True:
        i += 1
        yield i * i
```

Implementing cycle

https://docs.python.org/3/library/itertools.html#itertools.cycle

Generator Syntactic Sugar

- yield from
- Generator comprehensions
- Wrapping up a generator

Iterator Invalidation

- What happens when we modify something we're iterating over?

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

for number in numbers:
    if number == 3 or number == 4:
        numbers.remove(number)

print(numbers)
```

More interesting python topics

- https://python-course.eu

Feedback

Iterator Use Cases

- Most data structures provide in-built iterators
- Traversing non-linear data structures