# Advanced_python

August 26, 2020

# 1 Useful utilites and functionalities in Python for IN3120/IN4120

## 1.1 Python iterators

Every iterable datastructure has an `__iter__()` method.

```
In [1]: # To get the iterator we call the iter() function
        mylist = ["Welcome", "to", "search", "technology"]

        my_iterator = iter(mylist)
        for i in my_iterator:
            print(i)
```

```
Welcome
to
search
technology
```

```
In [2]: # The iterator can be exhausted
        for i in my_iterator: print(i)
```

Note: when you for loop through an iterable, python calls `iter` and `next` under the hood

**Manual traversal of iterator** we call next to get the next element from the iterator

```
In [3]: # Load a fresh iterator
        myit = iter(mylist)
        myit
```

```
Out[3]: <list_iterator at 0x7f9e301e3a90>
```

```
In [4]: # When the iterator is empty, StopIteration is raised
        # In for loops it is caught and makes the loop terminate
        current = next(myit)
        current
```

```
Out[4]: 'Welcome'
```

```
In [5]: next(myit)

Out[5]: 'to'

In [6]: next(myit)

Out[6]: 'search'

In [7]: next(myit)

Out[7]: 'technology'

In [8]: next(myit)

        ---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call last)

        <ipython-input-8-ec9104da3bd7> in <module>
    ----> 1 next(myit)


        StopIteration:


In [10]: # Problem: we don't want an error raised when the iterator is empty
         # Solution: add a default value to the next function
         # i.e. iter(<iterable>, <value-if-empty>)
         myit = iter(mylist)

In [11]: next(myit, "Iterator is empty")

Out[11]: 'Welcome'

In [12]: next(myit, "Iterator is empty")

Out[12]: 'to'

In [13]: next(myit, "Iterator is empty")

Out[13]: 'search'

In [14]: next(myit, "Iterator is empty")

Out[14]: 'technology'

In [15]: next(myit, "Iterator is empty")

Out[15]: 'Iterator is empty'
```

```
In [16]: # None is a typical default value
         # That makes it easy to check if the iterator is empty
         current = next(myit, None)
         if current:
             print("Iterator is not empty")
         else:
             print("Iterator is empty")
```

```
Iterator is empty
```

### 1.1.1 Generators

A generator is an iterator, but not all iterators are generators

```
In [17]: def my_iterator():
             return iter(range(5))

         def my_generator():
             for i in range(5):
                 yield i
             # Note: use of nonlocal variable
             for i in ["Welcome", "to", "search", "technology"]:
                 yield i

         my_generator
```

```
Out[17]: <function __main__.my_generator()>
```

```
In [18]: my_gen = my_generator()
         my_gen
```

```
Out[18]: <generator object my_generator at 0x7f9e30130190>
```

```
In [19]: next(my_gen)
```

```
Out[19]: 0
```

```
In [20]: for i in my_gen:
             print(i)
```

```
1
2
3
4
Welcome
to
search
technology
```

In [21]:
```python
# Extra: if you want to yield all elements from an iterable, use "yield from syntax"
def example():
    yield from range(1, 10)
    yield from range(10, 0, -1)

list(example())
```

Out[21]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

### 1.2 Zip function

Ever wanted to iterate from two iterables simultaneously?
Or perhaps wrap two lists into a list of pairs?
Or perhaps wrap n lists into a list of n-tuples?

In [22]:
```python
numbers = [5, 6, 7]
chars = ["b", "c", "d"]

# Basic method
for i in range(len(numbers)):
    print(numbers[i], chars[i])

# Equivalent with zip
for n, c in zip(numbers, chars):
    print(n, c)
```

```
5 b
6 c
7 d
5 b
6 c
7 d
```

In [23]:
```python
# Zip returns an iterator
it = zip(numbers, chars)
```

In [24]:
```python
# Note that the pairs are tuples and not lists
next(it)
```

Out[24]:
```
(5, 'b')
```

In [25]:
```python
list(zip(numbers, chars))
```

Out[25]:
```
[(5, 'b'), (6, 'c'), (7, 'd')]
```

### 1.3 List comprehensions

```
In [26]: # % is the modulo opreand: it returns the remainder of the left number divided by the
         def is_odd(n):
             return n % 2
```

```
In [27]: # say we want a list of squares of 0 through 9
         [i*i for i in range(10)]
```

```
Out[27]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [28]: # say we only want odd
         [i*i for i in range(10) if not is_odd(i)]
```

```
Out[28]: [0, 4, 16, 36, 64]
```

```
In [29]: # say if the number is odd, it is swapped out with 0
         [i*i if i % 2 == 0 else 0 for i in range(10)]
```

```
Out[29]: [0, 0, 4, 0, 16, 0, 36, 0, 64, 0]
```

## 2 syntax:

```
[(expression) for i in (iterable)]
[(expression) for i in (iterable) if (condition)]
[(expression) if (condition) else (expression) for i in (iterable)]
[(expression) for (iterable) in (nestediterable) for i in (iterable)]
```

### 2.1 Dict comprehensions

```
In [30]: {i:i.upper() for i in mylist}
```

```
Out[30]: {'Welcome': 'WELCOME',
          'to': 'TO',
          'search': 'SEARCH',
          'technology': 'TECHNOLOGY'}
```

Syntax:

```
{(key expression):(value expression) for i in (iterable)}
```

### 2.2 Generator comprehensions

Like list comprehension, but uses parentheses instead of brackets

```
In [31]: myit = (i*i if (i%2==0) else 0 for i in range(10))
```

### 2.3 Passing generator comprehensions

you dont need to put parentheses if the generator comprehension is the only argument for a function This lets us create comprehensions for any datastrucutre that can take iterables as inputs. Very elegant and pythonic

```
In [32]: sum(i*i for i in range(10))

Out[32]: 285

In [33]: set(i*i for i in range(10))

Out[33]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

### 2.4 Counters

Counter is a subclass of dict in python. It takes in an iterable of anything hashable and creates each unique element as key and its frequency as value

```
In [34]: from collections import Counter

         documents = [
             "I am a document",
             "I am an an an as",
             "I'm very very happy",
             "Ha ha ha ha"
         ]

         Counter(documents)

Out[34]: Counter({'I am a document': 1,
                  'I am an an an as': 1,
                  "I'm very very happy": 1,
                  'Ha ha ha ha': 1})

In [35]: # Normalization and tokenization
         tokenized_documents = [doc.lower().split() for doc in documents]
         tokenized_documents

Out[35]: [['i', 'am', 'a', 'document'],
          ['i', 'am', 'an', 'an', 'an', 'as'],
          ["i'm", 'very', 'very', 'happy'],
          ['ha', 'ha', 'ha', 'ha']]

In [36]: c1 = Counter(token for tokens in tokenized_documents for token in tokens)
         c1

Out[36]: Counter({'i': 2,
                  'am': 2,
                  'a': 1,
```

```
                 'document': 1,
                 'an': 3,
                 'as': 1,
                 "i'm": 1,
                 'very': 2,
                 'happy': 1,
                 'ha': 4})
```

```
In [37]: c2 = Counter([0,1,1,2,6,6,5,4, "i", "i"])
         c2
```

```
Out[37]: Counter({0: 1, 1: 2, 2: 1, 6: 2, 5: 1, 4: 1, 'i': 2})
```

```
In [38]: # Counters support additions, so that you can merge two counters
         c1 + c2
```

```
Out[38]: Counter({'i': 4,
                  'am': 2,
                  'a': 1,
                  'document': 1,
                  'an': 3,
                  'as': 1,
                  "i'm": 1,
                  'very': 2,
                  'happy': 1,
                  'ha': 4,
                  0: 1,
                  1: 2,
                  2: 1,
                  6: 2,
                  5: 1,
                  4: 1})
```

```
In [39]: mylist = [[1,2,3], [4,5,6], [7,8,9]]
```

```
In [40]: flatlist = []
         for sublist in mylist:
             for i in sublist:
                 flatlist.append(i)
```

```
In [41]: [i for sublist in mylist for i in sublist]
```

```
Out[41]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 2.5 Type hints

Type hints can be great for readability and is used quite a lot in the assignments.
It is a quite new python feature, but does not actually affect how the code runs

```
In [42]: def numerical_function(n: int, k: float) -> complex:
             return n + k + 1j

         def numerical_function_without_type_hints(n, k):
             return n + k + 1j
```

```
In [43]: # You can call the help function and see the type hints
         help(numerical_function)
         help(numerical_function_without_type_hints)
```

```
Help on function numerical_function in module __main__:

numerical_function(n: int, k: float) -> complex

Help on function numerical_function_without_type_hints in module __main__:

numerical_function_without_type_hints(n, k)
```

```
In [44]: # Type hints are only for the reader of the code. There is no type checking
         def numerical_function(n: int, k: float) -> complex:
             return "kødda"
```

```
In [45]: # For some typing, you will have to import auxiliary typing classes
         # Say if you'd like to know what is in the list that you are returning
         def list_of_letters() -> list:
             return ["a", "b", "c"]

         from typing import List
         def list_of_letters() -> List[str]:
             return ["a", "b", "c"]
```

### 2.6 Abstract classes

Unlike java, abstract classes are not built in the syntax of python.
However, there is a built in module that makes this possible

```
In [46]: from abc import ABC, abstractmethod
```

```
In [47]: class Shape(ABC):

             def foo(self):
                 return 42

             @abstractmethod
             def area(self):
                 pass
```

In equivalent java code:

```
abstract class Shape {
    public int foo() {
        return 42;
    }

    public abstract double area();
}
```

In [48]:
```python
class Square(Shape):
    def __init__(self, length):
        self.length = length

Square(5)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-48-a8d2e950dea7> in <module>
      3         self.length = length
      4
----> 5 Square(5)


TypeError: Can't instantiate abstract class Square with abstract methods area
```

In [49]:
```python
class Square(Shape):
    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

Square(5).area()
```

Out[49]: 25

In [ ]:

In [ ]: