BEGINNER

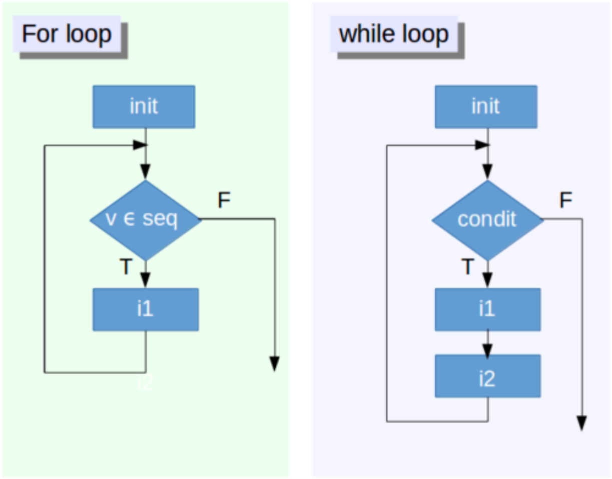
Loops

Concepts & Introduction

Context : Iterations

- Same code (over chunk of data for example)
- While loops
- For loops
- Generators
- Multiprocessing

For loops



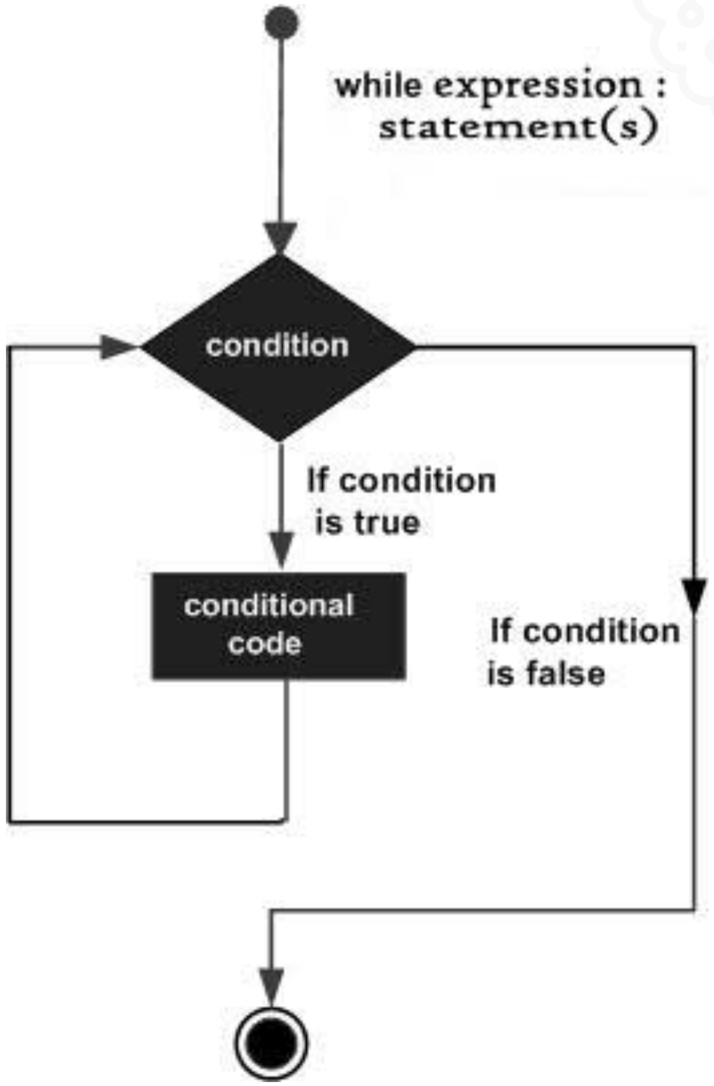
- For loops are the easiest
  - necessitates logic
  - number of iterations

- For loops are used to iterate
  - tuples
  - dictionaries
  - list

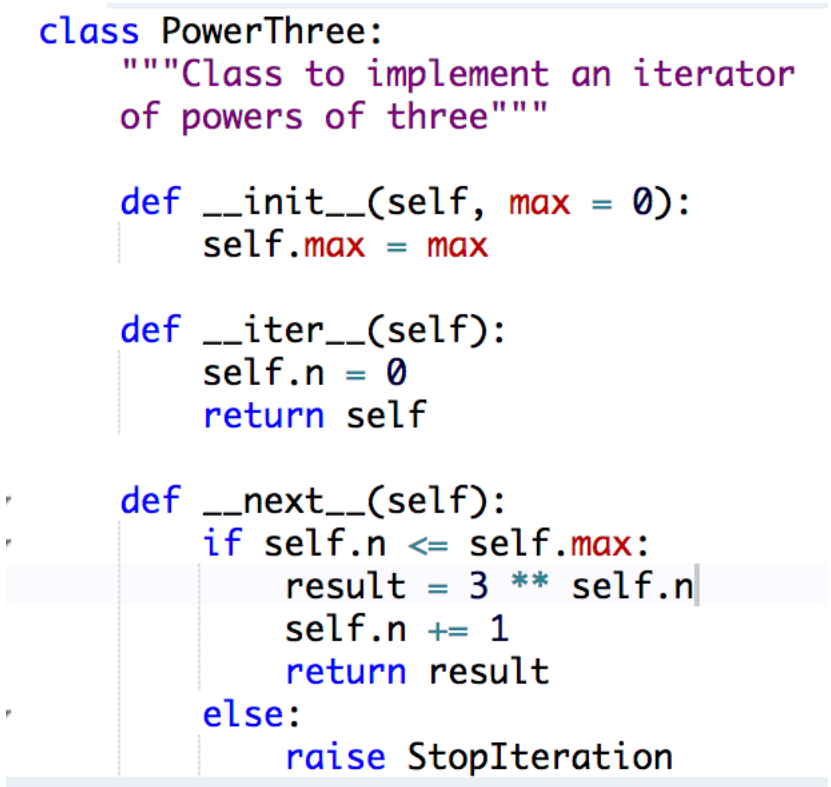
While loops

- While loops are not the easiest
  - necessitates logic to break the loop
  - no specified number of iterations

- While loops are used to iterate
  - generators
  - list



Iterators



```python
class PowerThree:
    """Class to implement an iterator
    of powers of three"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 3 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

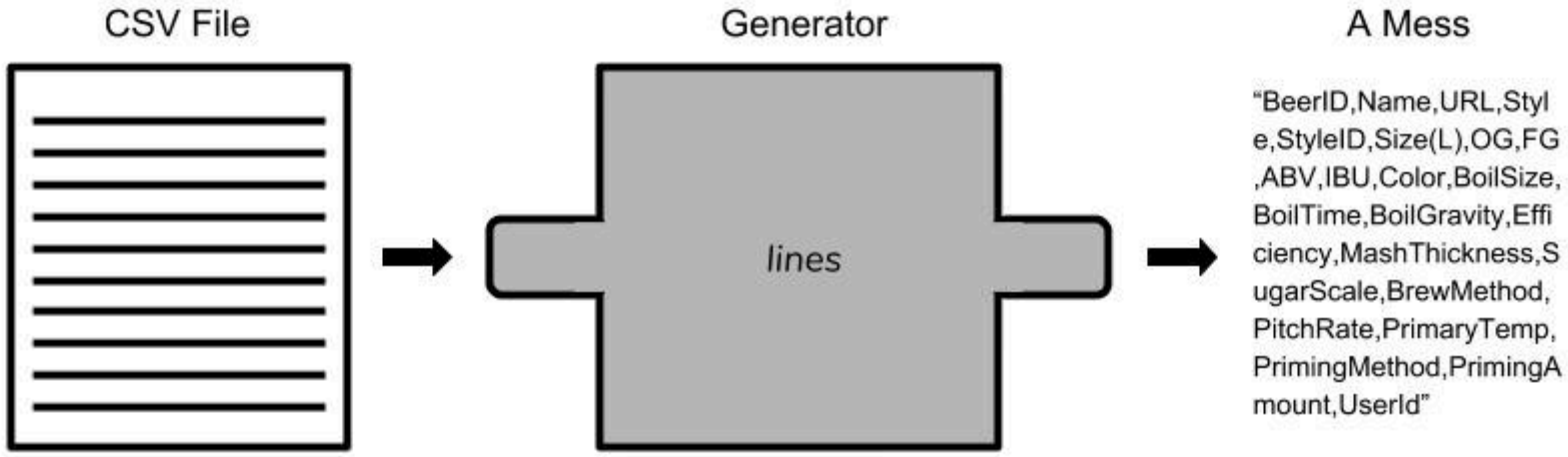
To build an iterator we have to implement the methods `__iter__()` and `__next__()`.

The `__iter__()` method returns the iterator object itself.

The `__next__()` method must return the next item in the sequence.

On reaching the end, and in subsequent calls, it must raise the error `StopIteration (and be handled)`

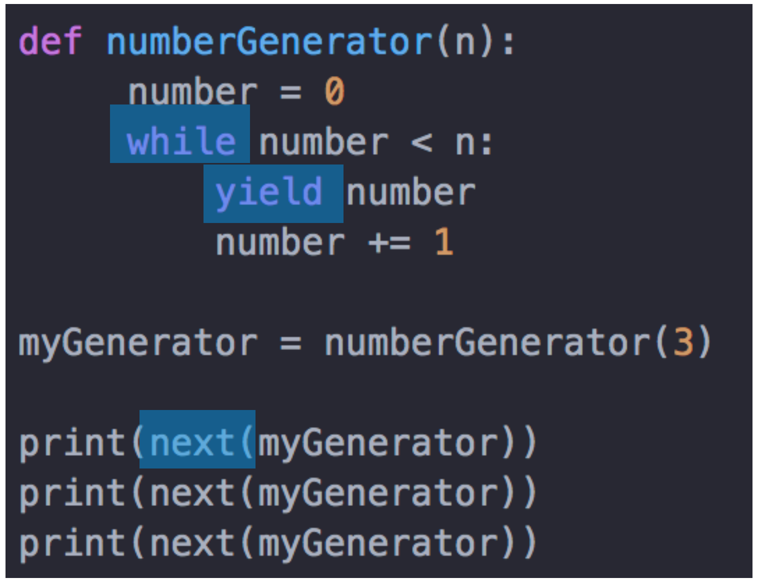
Generators



"BeerID,Name,URL,Style,StyleID,Size(L),OG,FG,ABV,IBU,Color,BoilSize,BoilTime,BoilGravity,Efficiency,MashThickness,SugarScale,BrewMethod,PitchRate,PrimaryTemp,PrimingMethod,PrimingAmount,UserId"

- Generator :
  - lower the impact on the memory
  - can increase the speed

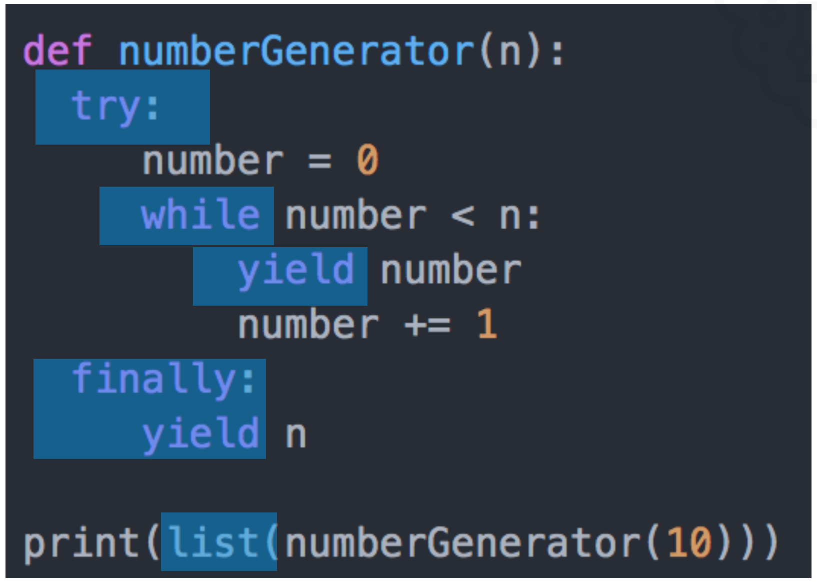
Generators



```python
def numberGenerator(n):
    number = 0
    while number < n:
        yield number
        number += 1

myGenerator = numberGenerator(3)

print(next(myGenerator))
print(next(myGenerator))
print(next(myGenerator))
```

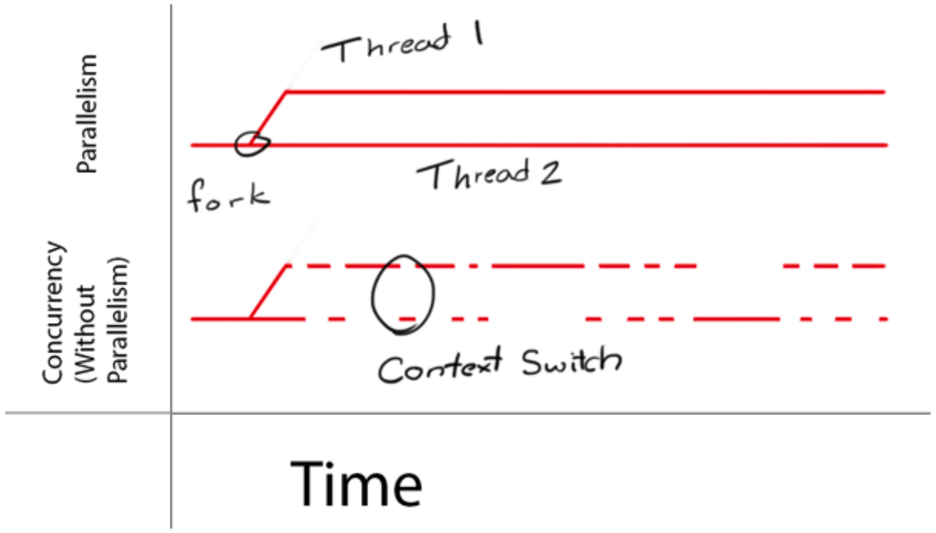


```python
def numberGenerator(n):
    try:
        number = 0
        while number < n:
            yield number
            number += 1
    finally:
        yield n

print(list(numberGenerator(10)))
```

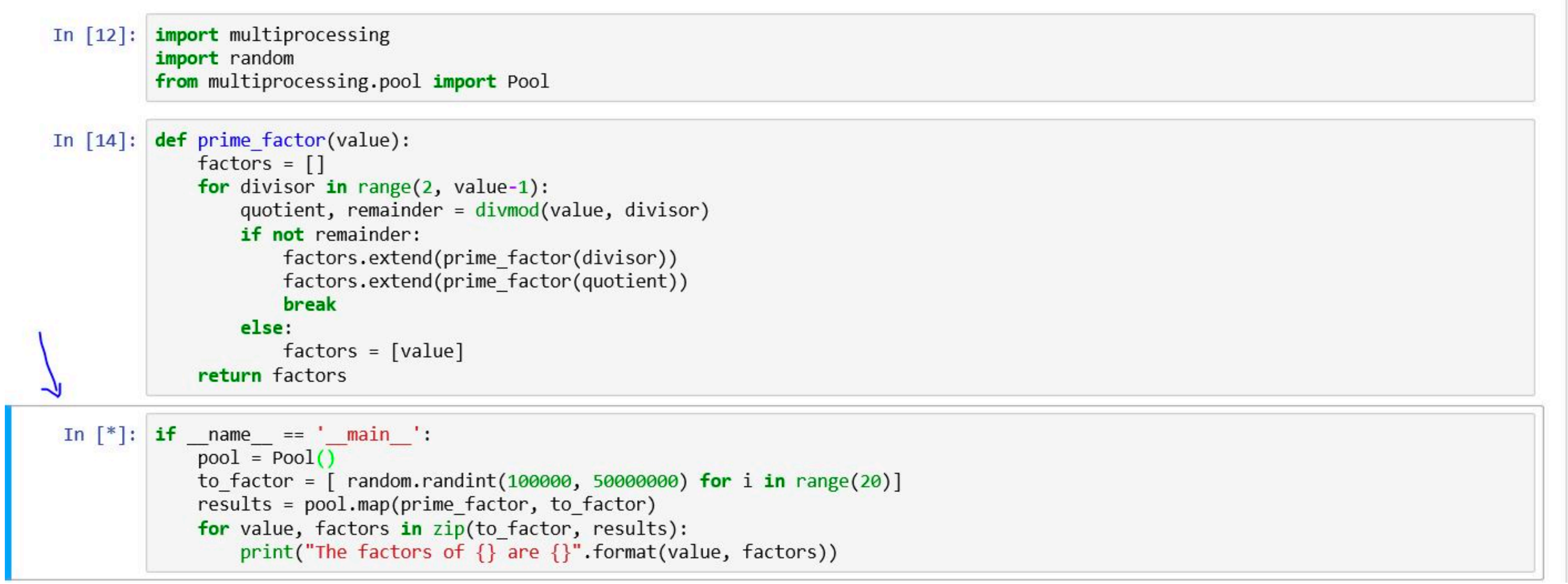
- Generator :
  - Slightly different from an object that supports iteration
  - One-time operation
    = (different from a list which you can iterate over & over)

Multiprocessing/Multithreading



- Please read : https://www.toptal.com/python/beginners-guide-to-concurrency-and-parallelism-in-python

Multiprocessing



```
In [12]: import multiprocessing
         import random
         from multiprocessing.pool import Pool

In [14]: def prime_factor(value):
             factors = []
             for divisor in range(2, value-1):
                 quotient, remainder = divmod(value, divisor)
                 if not remainder:
                     factors.extend(prime_factor(divisor))
                     factors.extend(prime_factor(quotient))
                     break
                 else:
                     factors = [value]
             return factors

In [*]: if __name__ == '__main__':
            pool = Pool()
            to_factor = [ random.randint(100000, 50000000) for i in range(20)]
            results = pool.map(prime_factor, to_factor)
            for value, factors in zip(to_factor, results):
                print("The factors of {} are {}".format(value, factors))
```

- A **process** is a collection of code, memory, data and other resources.
- A **thread** is a sequence of code that is executed within the scope of the **process**.
- You can (usually) have multiple **threads** executing concurrently within the same **process**

- *pool.apply(f, args):* is only executed in ONE of the workers of the pool. So ONE of the processes in the pool will do the job
- *pool.apply_async(f, args):* is also like Python's built-apply, except that the call returns immediately instead of waiting for the result.AsyncResult object is returned.You call method to retrieve the result of the function call. method blocks until the function is completed. In a sense : pool.apply(func, args), is equivalent pool.apply_async(func, args, kwargs).get()
- *pool.map(f,args):* pool.map(func, args) applies the same function to many arguments

https://stackoverflow.com/questions/8533318/multiprocessing-pool-when-to-use-apply-apply-async-or-map

stack overflow