Generators, iterators

- __iter__, __next__
- yield
- generator expression
- measuring memory usage

Iterable & Iterator

Python shell

```
> L = ['a', 'b', 'c']
> type(L)
| <class 'list'>
> it = L.__iter__()
> type(it)
| <class 'list_iterator'>
> it.__next__()
| 'a'
> it.__next__()
| 'b'
> it.__next__()
| 'c'
> it.__next__()
| StopIteration   # Exception
```

Python shell

```
> L = ['a', 'b', 'c']
> it = iter(L)   # calls L.__iter__()
> next(it)       # calls it.__next__()
| 'a'
> next(it)
| 'b'
> next(it)
| 'c'
> next(it)
| StopIteration
```

iterator ≈ pointer into list

`['a', 'b', 'c']`

- Lists are iterable (must support `__iter__`)
- `iter` returns an iterator (must support `__next__`)

Some iterables in Python: string, list, set, tuple, dict, range, enumerate, zip, map, reversed

Iterator

- next(*iterator_object*) returns the next element from the iterator, by calling the *iterator_object*`.__next__()`. If no more elements to report, raises exception `StopIteration`
- next(*iterator_object*, *default*) returns *default* when no more elements are available (no exception is raised)
- for-loops and list comprehensions require iterable objects
  `for x in range(5):` and `[2**x for x in range(5)]`
- The iterator concept is also central to Java and C++

for loop

Python shell

```
> for x in ['a', 'b', 'c']:
      print(x)
| a
| b
| c
```

result of next on iterator

iterable object (can call iter on it to generate an iterator)

≡

Python shell

```
> L = ['a', 'b', 'c']
> it = iter(L)
> while True:
      try:
          x = next(it)
      except StopIteration:
          break
      print(x)
| a
| b
| c
```

8.3. The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the `else` clause, if present, is executed, and the loop terminates.

docs.python.org/3/reference/compound_stmts.html#the-for-statement

`for` loop over changing iterable

Changing (extending) the list while scanning
The iterator over a list is just an index into the list

Python shell

```
> L = [1, 2]
> for x in L:
      print(x, L)
      L.append(x + 2)
| 1 [1, 2]
| 2 [1, 2, 3]
| 3 [1, 2, 3, 4]
| 4 [1, 2, 3, 4, 5]
| 5 [1, 2, 3, 4, 5, 6]
...
```

Python shell

```
> L = [1, 2]
> for x in L:
      print(x, L)
      L[:0] = [L[0] - 2, L[0] - 1]
| 1 [1,2]
| 0 [-1,0,1,2]
| -1 [-3,-2,-1,0,1,2]
| -2 [-5,-4,-3,-2,-1,0,1,2]
| -3 [-7,-6,-5,-4,-3,-2,-1,0,1,2]
...
```

range

Python shell

```
> r = range(1, 6)   # 1,2,3,4,5
> type(r)
| <class 'range'>
> it = iter(r)
> type(it)
| <class 'range_iterator'>
> next(it)
| 1
> next(it)
| 2
> for x in it:
      print(x)
| 3
| 4
| 5
```

iterable expected but got iterator ?

Python shell

```
> it
| <range_iterator object at 0x03E7FFC8>
> iter(it)
| <range_iterator object at 0x03E7FFC8>
> it is iter(it)
| True
```

Calling `iter` on a `range_iterator` just returns the iterator itself, i.e. can use the iterator wherever an iterable is expected

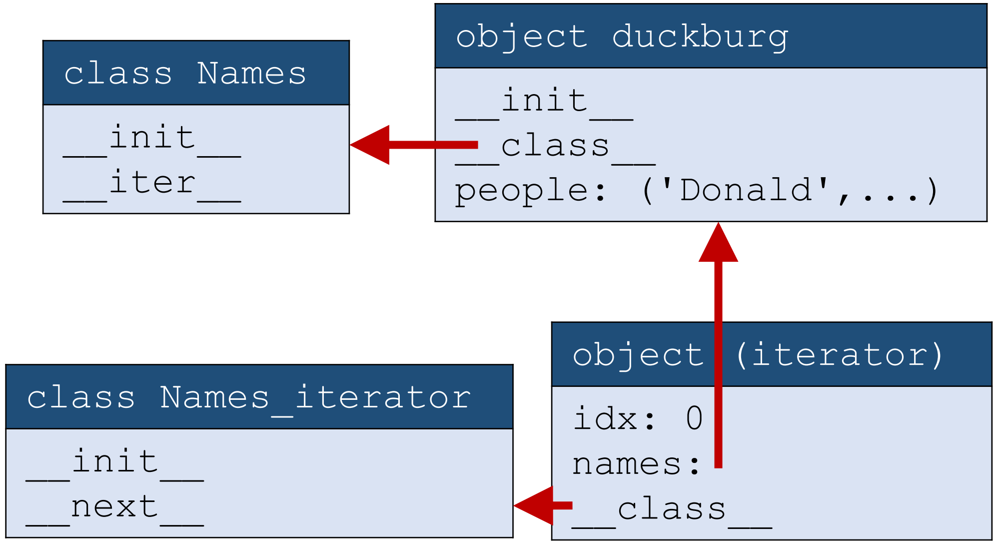
Creating an interable class

names.py

```
class Names:
    def __init__(self, *arg):
        self.people = arg
    def __iter__(self):
        return Names_iterator(self)
class Names_iterator:
    def __init__(self, names):
        self.idx = 0
        self.names = names
    def __next__(self):
        if self.idx >= len(self.names.people):
            raise StopIteration
        self.idx += 1
        return self.names.people[self.idx - 1]
duckburg = Names('Donald', 'Goofy', 'Mickey', 'Minnie')
for name in duckburg:
    print(name)
```

Python shell

```
| Donald
| Goofy
| Mickey
| Minnie
```



An infinite iterable

infinite_range.py

```python
class infinite_range:
    def __init__(self, start=0, step=1):
        self.start = start
        self.step = step

    def __iter__(self):
        return infinite_range_iterator(self)

class infinite_range_iterator:
    def __init__(self, inf_range):
        self.range = inf_range
        self.current = self.range.start

    def __next__(self):
        value = self.current
        self.current += self.range.step
        return value

    def __iter__(self):   # make iterator iterable
        return self
```

Python shell

```
> r = infinite_range(42, -3)
> it = iter(r)
> for idx, value in zip(range(5), it):
      print(idx, value)
| 0 42
| 1 39
| 2 36
| 3 33
| 4 30
> for idx, value in zip(range(5), it):
      print(idx, value)
| 0 27
| 1 24
| 2 21
| 3 18
| 4 15
> print(sum(r))   # don't do this
| (runs forever)
```

`sum` and `zip` take iterables

(`zip` stops when shortest iterable is exhausted)

Creating an iterable class (iterable = iterator)

my_range.py

```
class my_range:
    def __init__(self, start, end, step):
        self.start = start
        self.end = end
        self.step = step
        self.x = start

    def __iter__(self):
        return self  # self also iterator

    def __next__(self):
        if self.x >= self.end:
            raise StopIteration
        answer = self.x
        self.x += self.step
        return answer

r = my_range(1.5, 2.0, 0.1)
```

Python shell

```
> list(r)
| [1.5, 1.6,
  1.7000000000000002,
  1.8000000000000003,
  1.9000000000000004]
> list(r)
| []
```

- Note that objects act both as an iterable and an iterator
- This e.g. also applies to `zip` objects
- Can only iterate over a `my_range` once

itertools

Function	Description
`count(start, step)`	Inifinite sequence: `start`, `stat` + `step`, …
`cycle(seq)`	Infinite repeats of the elements from `seq`
`repeat(value[, times])`	Infinite repeats of `value` or `times` repeats
`chain(seq0,...,seqk)`	Concatenate sequences
`starmap(func, seq)`	`func(*seq[0]), func(*seq[1]),` …
`permutations(seq)`	Genereate all possible permutations of `seq`
`islice(seq, start, stop, step)`	Create a slice of `seq`
…	…

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

Example : Java iterators

vector-iterator.java

```
import java.util.Vector;
import java.util.Iterator;

class IteratorTest {
   public static void main(String[] args) {
       Vector<Integer> a = new Vector<Integer>();
       a.add(7);
       a.add(42);
       // "C" for-loop & get method
       for (int i=0; i<a.size(); i++)
          System.out.println(a.get(i));
       // iterator
       for (Iterator it = a.iterator(); it.hasNext(); )
          System.out.println(it.next());
       // for-each loop - syntax sugar since Java 5
       for (Integer e : a)
          System.out.println(e);
   }
}
```

In Java iteration does not stop using exceptions, but instead the iterator can be tested if it is at the end of the iterable

Example : C++ iterators

vector-iterator.cpp

```
#include <iostream>
#include <vector>
int main() {
  // Vector is part of STL (Standard Template Library)
  std::vector<int> A = {20, 23, 26};
  // "C" indexing - since C++98
  for (int i = 0; i < A.size(); i++)
    std::cout << A[i] << std::endl;
  // iterator - since C++98
  for (std::vector<int>::iterator it = A.begin(); it != A.end(); ++it)
    std::cout << *it << std:: endl;
  // "auto" iterator - since C++11
  for (auto it = A.begin(); it != A.end(); ++it)
    std::cout << *it << std:: endl;
  // Range-based for-loop - since C++11
  for (auto e : A)
    std::cout << e << std:: endl;
}
```

In C++ iterators can be tested if they reach the end of the iterable

move iterator to next element

Generators

Generator expressions

Python shell

```
> [x ** 2 for x in range(5)]   # list comprehension
| [0, 1, 4, 9, 16]  # list
> (x ** 2 for x in range(3))   # generator expression
| <generator object <genexpr> at 0x03D9F8A0>
> o = (x ** 2 for x in range(3))
> next(o)
| 0
> next(o)
| 1
> next(o)
| 4
> next(o)
| StopIteration
```

- A generator expression
  `(... for x in ...)`
  looks like a list comprehension, except square brackets are replaced by parenthesis
- Is an iterable and iterator, that uses less memory than a list comprehension
- computation is done *lazily*, i.e. first when needed

https://docs.python.org/3/reference/expressions.html#generator-expressions

Nested generator expressions

Python shell

```
> squares = (x ** 2 for x in range(1, 6))  # generator expression
> ratios = (1 / y for y in squares)  # generator expression
> ratios
| <generator object <genexpr> at 0x031FC230>
> next(ratios)
| 1.0
> next(ratios)
| 0.25
> print(list(ratios))
| [0.1111111111111111, 0.0625, 0.04]  # remaining 3
```

- Each fraction is first computed when requested by `next(ratios)` (implicitly called repeatedly in `list(ratios)`)
- The next value of `squares` is first computed when needed by `ratios`

Generator expressions as function arguments

Python shell

```
> doubles = (x * 2 for x in range(1, 6))
> sum(doubles)   # sum takes an iterable
| 30
> sum((x * 2 for x in range(1, 6)))
| 30
> sum(x * 2 for x in range(1, 6))   # one pair of parenthesis omitted
| 30
```

- Python allows to omit a pair of parenthesis when a generator expression is the only argument to a function

`f(... for x in ...)` ≡ `f((... for x in ...))`

PEP 289 – Generator Expressions

Generator functions

```
two.py
def two():
    yield 1
    yield 2
```

```
Python shell
> two()
| <generator object two at 0x03629510>
> t = two()
> next(t)
| 1
> next(t)
| 2
> next(t)
| StopIteration
```

- A *generator function* contains one or more `yield` statements
- Python automatically makes a call to a generator function into an iterable and iterator (provides `__iter__` and `__next__`)
- Calling a generator function returns a *generator object*
- Whenever `next` is called on a generator object, the excuting of the function continues until the next `yield` ***exp*** and the value of *exp* is returned as a result of `next`
- Reaching the end of the function or a return statement, will raise `StopIteration`
- Once consumed, can't be reused

https://docs.python.org/3/reference/expressions.html#yield-expressions

Generator functions (II)

my_generator.py

```python
def my_generator(n):
    yield 'Start'
    for i in range(n):
        yield chr(ord('A') + i)
    yield 'Done'
```

Python shell

```
> g = my_generator(3)
> print(g)
| <generator object my_generator at 0x03E2F6F0>
> print(list(g))
| ['Start', 'A', 'B', 'C', 'Done']
> print(list(g))   # generator object g exhausted
| []
> print(*my_generator(5))   # * takes an iterable (PEP 448)
| Start A B C D E Done
```

Generator functions (III)

my_range_generator.py

```
def my_range(start, end, step):
    x = start
    while x < end:
        yield x
        x += step
```

Python shell

```
> list(my_range(1.5, 2.0, 0.1))
| [1.5, 1.6, 1.7000000000000002, 1.8000000000000003, 1.9000000000000004]
```

Pipelining generators

Python shell

```
> def squares(seq):       # seq should be an iterable object
      for x in seq:       # use iterator to run through seq
          yield x ** 2    # generator
> list(squares(range(5)))
| [0, 1, 4, 9, 16]
> list(squares(squares(range(5))))   # pipelining generators
| [0, 1, 16, 81, 256]
> sum(squares(squares(range(100000000))))   # pipelining generators
| 19999999500000003333333333333330000000
> sum((x ** 2) ** 2 for x in range(100000000))   # generator expression
| 19999999500000003333333333333330000000
> sum([(x ** 2) ** 2 for x in range(100000000)])   # list comprehension
| MemoryError  # when using a 32-bit version of Python, limited to 2 GB
```

`yield` vs `yield from`

Python shell

```
> def g():
      yield 1
      yield [2, 3, 4]
      yield 5
> list(g())
| [1, [2, 3, 4], 5]
```

Python shell

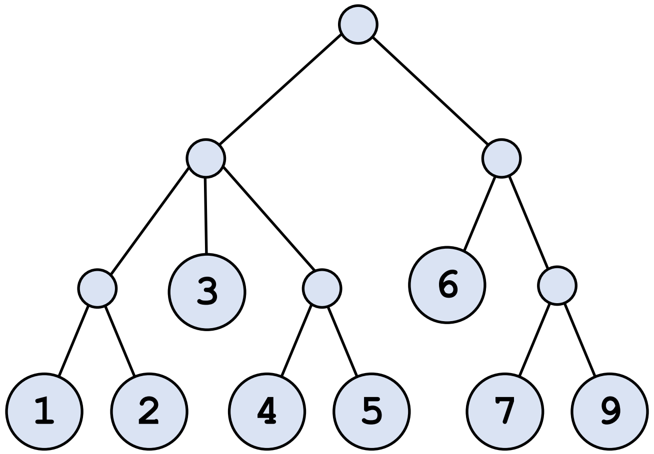
```
> def g():
      yield 1
      yield from [2, 3, 4]
      yield 5
> list(g())
| [1, 2, 3, 4, 5]
```

- `yield from` available since Python 3.3
- `yield from` *`exp`* ≈ `for x in` *`exp`*`: yield x`

Recursive `yield from`

Python shell

```
> def traverse(T):   # recursive generator
      if isinstance(T, tuple):
          for child in T:
              yield from traverse(child)
      else:
          yield T
> T = (((1, 2), 3, (4, 5)), (6, (7, 9)))
> traverse(T)
| <generator object traverse at 0x03279F30>
> list(traverse(T))
| [1, 2, 3, 4, 5, 6, 7, 9]
```



Making objects iterable using `yield`

vector2D.py

```
class vector2D:
    def __init__(self, x_value, y_value):
        self.x = x_value
        self.y = y_value
    def __iter__(self):  # generator
        yield self.x
        yield self.y
    def __iter__(self):  # alternative generator
        yield from (self.x, self.y)
v = vector2D(5, 7)
print(list(v))
print(tuple(v))
print(set(v))
```

Python shell

```
[5, 7]
(5, 7)
{5, 7}
```

Generators vs iterators

- Iterators can often be reused (can copy the current state)
- Generators cannot be reused (only if a new generator is created, starting over again)

- David Beazley's tutorial on
  "*Generators: The Final Frontier*", PyCon 2014 (3:50:54)
  Throughout advanced discussion of generators, e.g. how to use `.send` method to implement coroutines
  https://www.youtube.com/watch?v=D1twn9kLmYg

Measuring memory usage

Measuring memory usage (memory profiling)

- Macro level:

  Task Manager (Windows)
  Activity Monitor (Mac)
  top (Linux)

- Variable level:

  `getsizeof` from `sys` module

- Detailed overview:

  Module `memory_profiler`

  Allows detailed space usage of the code line-by-line (using @profile function decorator) or a plot of total space usage over time

  pip install memory-profiler

Python shell

```
> import sys
> sys.getsizeof(42)
| 14   # size of the integer 42 is 14 bytes
> sys.getsizeof(42 ** 42)
| 44   # the size increases with value
> sys.getsizeof('42')
| 27   # size of a string
> import numpy as np
> sys.getsizeof(np.array(range(100), dtype='int32'))
| 448   # also works on Numpy arrays
> squares = [x ** 2 for x in range(1000000)]
> sys.getsizeof(squares)
| 4348736
> g = (x ** 2 for x in range(1000000))
> sys.getsizeof(g)
| 64
```

Module memory-profiler

pypi.org/project/memory-profiler/

memory_usage.py

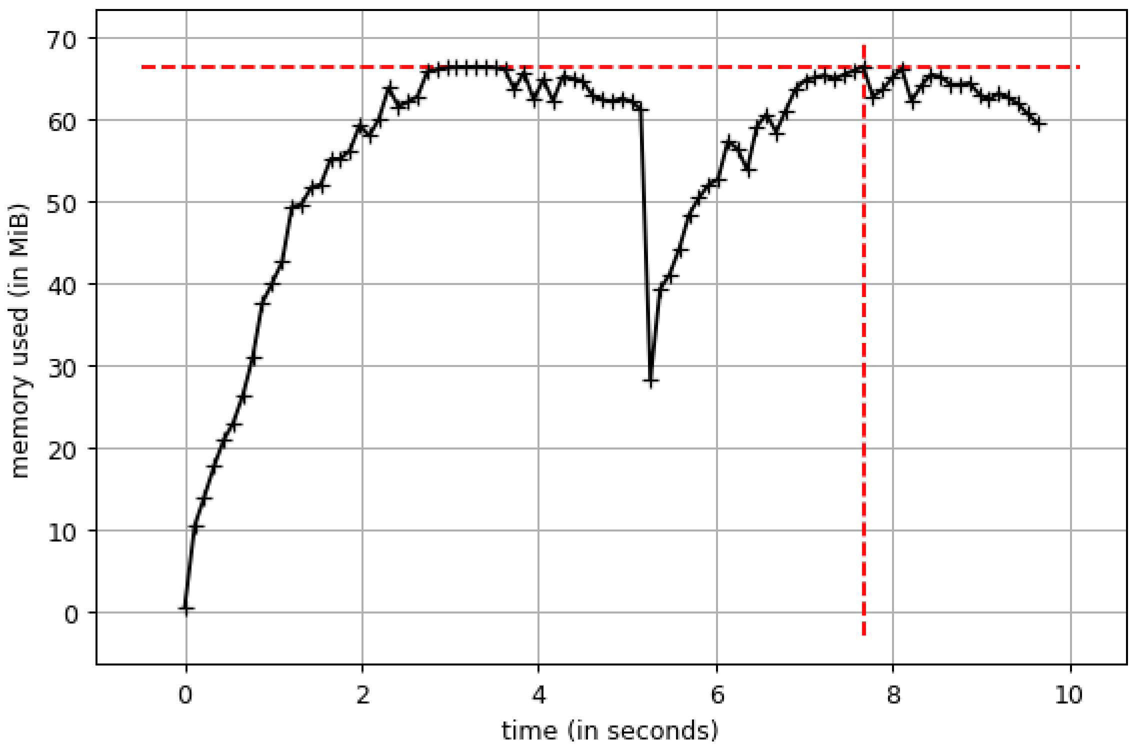
```
from memory_profiler import profile

@profile  # prints new statistics for each call
def use_memory():
    s = 0
    x = list(range(20_000_000))
    s += sum(x)
    y = list(range(10_000_000))
    s += sum(x)

use_memory()
```

Python Shell

```
| Filename: C:/.../memory_usage.py
|
| Line #   Mem usage   Increment   Line Contents
| ================================================
|      3   32.0 MiB   32.0 MiB  @profile
|      4                        def use_memory():
|      5   32.0 MiB    0.0 MiB      s = 0
|      6  415.9 MiB  383.9 MiB      x = list(range(20_000_000))
|      7  415.9 MiB    0.0 MiB      s += sum(x)
|      8  607.8 MiB  191.9 MiB      y = list(range(10_000_000))
|      9  607.8 MiB    0.0 MiB      s += sum(x)
```



memory_sin_usage.py

```
from math import sin, pi

for a in range(1000):
    x = list(range(int(1000000 * sin(pi * a / 250))))
```

Windows Shell

```
> pip install memory-profiler
> mprof run memory_sin_usage.py
| mprof: Sampling memory every 0.1s
| running as a Python program...
> mprof plot
```