Generators, iterators

- yield
- generator expression
- __iter__, __next__

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
```

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

Some iterables in Python: 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 be report raise 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++.

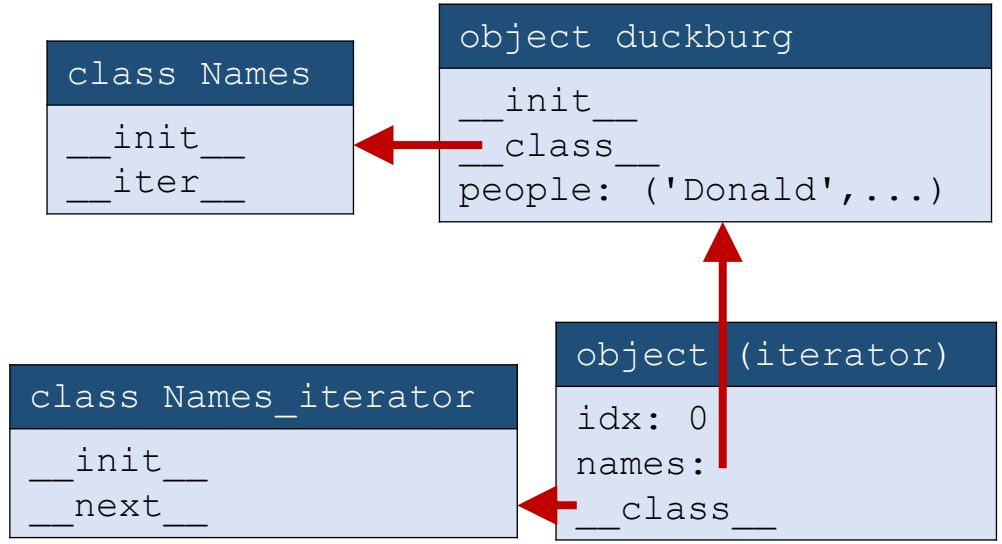
Creating an interable class

Python shell

```
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
```



Creating an interable class (iterable = iterator)

Python shell

```
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]
```

- Note that objects act both as an iterable and an iterator

Generator expressions

Python shell

```
> [x**2 for x in range(3)]  # 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 iterator, that uses less space than a list comprehension

https://docs.python.org/3/reference/expressions.html#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 the function into an iterator (provides `__iter__` and `__next__`)
- Calling a generator 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`

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

Generator functions (II)

my_generator.py

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

Python shell

```
> g = my_generator(3)
| <generator object two at 0x03629510>
> print(g)
| <generator object my_generator at
  0x03E2F6F0>
> print(list(g))
| ['Start', 'A', 'B', 'C', '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]
```

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

Making objects iterable using `yield`

my_generator.py

```
class vector2D:
    def __init__(self, x_value, y_value):
        self.x = x_value
        self.y = y_value
    def __iter__(self):
        yield self.x
        yield 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 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