04 Lists and Iterators

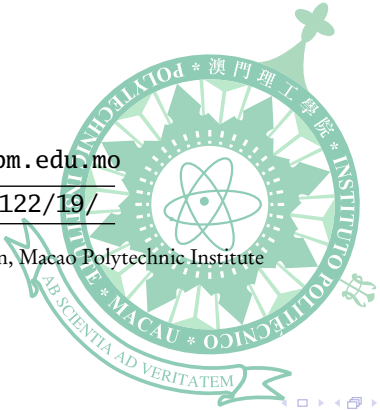
Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

January 18, 2019



Outline

Lists

- A `list` is a sequence of zero or more object references.

```
>>> ls = [0,1,2,3,[200,300,400],4]
>>> ls
[0, 1, 2, 3, [200, 300, 400], 4]
```

```
>>> ss = list('hello')
>>> ss
['h', 'e', 'l', 'l', 'o']
```

- Lists support the same indexing and slicing syntax as strings. This makes it easy to extract items from a list.

```
>>> ls[4][0:2]
[200, 300]
```

```
>>> ss[1] = 'E'
>>> ss
['h', 'E', 'l', 'l', 'o']
```

```
>>> ss[-3:-1] = ['L']
>>> ss
['h', 'E', 'L', 'o']
```

- Unlike strings, lists are mutable, so we can replace and delete any of their items.
- It is also possible to insert, replace, and delete slices of lists.
- Lists can be created by list literals, and the `list()` constructor.

Common List Operations

- The length of a list is obtained by the `len()` function, which calls the *__len__()* method.
- Whether an item is in a list can be checked by the `in` and `not in` operators, both depending on the *__contains__()* method.

```
>>> ps = [2,3,5,7,11]
>>> 9 not in ps
True
```

```
>>> 7 in ps
True
```

- A new item x can be appended to the end of a list l by the $l.append(x)$ method.
- An item x can be inserted at the index i of a list l by the $l.insert(i,\ x)$ method.
- We concatenate two lists by the (+) operator, and repeat a list by the (∗) operator.
- We can also extend and repeat a list in-place by the (+=) and (∗=) operators, respectively.

```
>>> ls = [1,2,3]
>>> ls*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> ls *= 2
>>> ls
[1, 2, 3, 1, 2, 3]
```

Unpacking Lists

- A list, in fact any iterable collection, can be unpacked using the unpacking operator (*).
- There are two or more variables on the left-hand side of such an assignment, one of which is preceded by * (starred).

```
>>> first, *mid, sec_last, last = [1,2,3,4,5,6]
```

- List items are assigned to the variables respectively, with all those left over assigned to the starred variable.

```
>>> first, mid, sec_last, last
(1, [2, 3, 4], 5, 6)
```

- We can also unpack a list to supply multiple arguments to a function from the list. The length of the list must match the number of function parameters.

```
>>> ls = [2, 20, 3]
>>> list(range(*ls))
[2, 5, 8, 11, 14, 17]
```

```
>>> ss = [1,2,3,4]
>>> abs(Vec(*ss[-2:]))
5.0
```

List Comprehensions

- Small lists are often created using list literals, but longer lists are usually created programmatically.
- Suppose, we want to produce a list of the leap years in a given range.

```
short_leaps = []
for year in range(1900, 1940):
    if year%4 == 0 and year%100 != 0 or year%400 == 0:
        short_leaps.append('{:02d}'.format(year%100))
```

- A list comprehension is an *expression* and a *loop* with an optional *condition* enclosed in brackets, where the loop is used to generate items for the list.

```
>>> ['{:02d}'.format(year%100) for year in range(1900, 1940)
     if year%4 == 0 and year%100 != 0 or year%400 == 0]
['04', '08', '12', '16', '20', '24', '28', '32', '36']
```

- The condition is used to filter out unwanted items.
- The expression is used to apply additional processing to the selected items.

Iterators

- An iterator is a process of returning items, one at a time, through the *__next__()* special method. An object provides its iterator through the *__iter__()* special method.

```
class MyIter:
    def __init__(self, n): self.n, self.i = n, 0
    def __next__(self):
        if self.i >= self.n:
            raise StopIteration
        x, self.i = self.i, self.i+1
        return x
    def __iter__(self): return self
```

- When no item can be returned, the iterator raises the *StopIteration* exception.

```
>>> list(MyIter(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Iterables

- An iterable collection, simply called an iterable, is an abstract collection that can iterates the elements through iterators.

```
class MyRange:
    def __init__(self, n): self.n = n
    def __iter__(self): return MyIter(self.n)
```

- While an iterator is a one time process, an iterable is more like a collection that can provide multiple iterators to iterate the elements independently at the same time.

```
>>> i = MyIter(3)
>>> [(x,y) for x in i for y in i]
[(0, 1), (0, 2)]
>>> r = MyRange(3)
>>> [(x,y) for x in r for y in r]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Generator Functions

- Python provides generator functions to simplify the writing of iterators. That is, a generator function returns an iterator.
- A generator function generates all the items of the iterator in one place.
- A generator returns an item in a `yield` statement and pauses, when the next item is requested, the generator resumes execution until the next `yield` statement.

```
def MyGen(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

- When a generator finishes, it raises the *StopIteration* exception automatically.

```
>>> list(MyGen(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Generator Expression

- A generator can also be specified by a generator expression, which is almost identical to a list comprehension, without the enclosing brackets.

```
>>> list(2*x+1 for x in range(10) if x%4 != 0)
[3, 5, 7, 11, 13, 15, 19]
```

- In a generator function, we can also yield a series of items from an iterator by the `yield from` statement. Now, an iterable can be defined much simpler by implementing the `__iter__()` special method as a generator function.

```
class MySquares:
    def __init__(self, n): self.n = n
    def __iter__(self):
        yield from (x*x for x in MyGen(self.n))
>>> r = MySquares(3)
>>> [(x,y) for x in r for y in r]
[(0, 0), (0, 1), (0, 4), (1, 0), (1, 1), (1, 4), (4, 0), (4, 1), (4, 4)]
```

Composing Iterators

- Zipping two iterators is to pair the corresponding items from each iterator to form an iterator of pairs. This function can help iterate though two iterators simultaneously.

```
def zip_iter(xs, ys):
    ixs, iys = iter(xs), iter(ys)
    while True:
        yield (next(ixs), next(iys))
```

```
def inf_from(i):
    while True:
        yield i
        i += 1
```

- We use the *zip_iter* function to number some string items.

```
>>> list(zip_iter(inf_from(1), ['Apple', 'Banana', 'Watermelon']))
[(1, 'Apple'), (2, 'Banana'), (3, 'Watermelon')]
```

- An infinite iterator can also be defined. The *take* function is used to take the first *n* items.

```
def take(xs, n):
    yield from (x for (i, x) in zip_iter(range(n), xs))
```

Sieve of Eratosthenes

- The Sieve of Eratosthenes is a method to generate prime numbers.
- The method works recursively on a sequence of natural numbers, starting from 2.
- The first number of the sequence is regarded as a prime number, the rest of the sequence is then filtered by this first number.
- We recursively apply the Sieve of Eratosthenes method to the filtered rest.

```
def sieve(s):
    i = iter(s)
    p = next(i)
    yield p
    yield from sieve(x for x in i if x%p != 0)
```

- The first 16 primes are obtained as the following.

```
>>> list(take(sieve(inf_from(2)), 16))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```