- 可迭代的对象、迭代器和生成器
  - 迭代是数据处理的基石，扫描内存中放不下的数据集时，我们要找到一种惰性获取数据项的方式，即按需一次获取一个数据项。这就是迭代器模式（Iterator pattern）。
  - 所有生成器都是迭代器，因为生成器完全实现了迭代器接口。
  - yield 关键字。3这个关键字用于构建生成器（generator），其作用与迭代器一样。
  - 序列可以迭代的原因：iter函数
    - 解释器需要迭代对象 x 时，会自动调用 iter(x)。
    - 内置的 iter 函数有以下作用
      - (1) 检查对象是否实现了 __iter__ 方法，如果实现了就调用它，获取一个迭代器。
      - (2) 如果没有实现 __iter__ 方法，但是实现了 __getitem__ 方法，Python 会创建一个迭代器，尝试按顺序（从索引 0 开始）获取元素。
      - (3) 如果尝试失败，Python 抛出 TypeError 异常，通常会提示"C object is not iterable"（C 对象不可迭代），其中 C 是目标对象所属的类。
    - 任何 Python 序列都可迭代的原因是，它们都实现了 __getitem__ 方法。
    - 这是鸭子类型（duck typing）的极端形式：不仅要实现特殊的 __iter__ 方法，还要实现 __getitem__ 方法，而且 __getitem__ 方法的参数是从 0 开始的整数（int），这样才认为对象是可迭代的。
      - 在白鹅类型（goose-typing）理论中，可迭代对象的定义简单一些，不过没那么灵活：如果实现了 __iter__ 方法，那么就认为对象是可迭代的。
      - 此时，不需要创建子类，也不用注册，因为 abc.Iterable 类实现了 __subclasshook__ 方法
    - 从 Python 3.4 开始，检查对象 x 能否迭代，最准确的方法是：调用 iter(x) 函数，如果不可迭代，再处理 TypeError 异常。
      - iter(x) 函数会考虑到遗留的 __getitem__ 方法，而 abc.Iterable 类则不考虑。
      - 如果除了抛出TypeError 异常之外还要做进一步的处理，可以使用 try/except 块，而无需显式检查。
      - 如果要保存对象，等以后再迭代，或许可以显式检查，因为这种情况可能需要尽早捕获错误。
  - 可迭代的对象与迭代器的对比
    - 可迭代的对象
      - 使用 iter 内置函数可以获取迭代器的对象。如果对象实现了能返回迭代器的 __iter__ 方法，那么对象就是可迭代的。
      - 序列都可以迭代；实现了 __getitem__ 方法，而且其参数是从零开始的索引，这种对象也可以迭代。
      - Python 从可迭代的对象中获取迭代器。
    - 迭代器
      - 迭代器是这样的对象：实现了无参数的 __next__ 方法，返回序列中的下一个元素；如果没有元素了，那么抛出 StopIteration 异常。
      - Python 中的迭代器还实现了 __iter__ 方法，因此迭代器也可以迭代。
    - 标准的迭代器接口
      - 两个方法
        - __next__ 返回下一个可用的元素，如果没有元素了，抛出 StopIteration 异常。
        - __iter__ 返回 self，以便在应该使用可迭代对象的地方使用迭代器，例如在 for 循环中。
      - 这个接口在 collections.abc.Iterator 抽象基类中制定。
        - 这个类定义了 __next__ 抽象方法，而且继承自 Iterable 类；__iter__ 抽象方法则在 Iterable 类中定义。
        - 
        - Iterator 抽象基类实现 __iter__ 方法的方式是返回实例本身（return self）。这样，在需要可迭代对象的地方可以使用迭代器。
    - 因为迭代器只需 __next__ 和 __iter__ 两个方法，所以除了调用 next() 方法，以及捕获 StopIteration 异常之外，没有办法检查是否还有遗留的元素。此外，也没有办法"还原"迭代器。
      - 如果想再次迭代，那就要调用 iter(...)，传入之前构建迭代器的可迭代对象。
    - 检查对象 x 是否为迭代器最好的方式是调用 isinstance(x, abc.Iterator)。
      - 得益于 Iterator.__subclasshook__ 方法，即使对象 x 所属的类不是 Iterator 类的真实子类或虚拟子类，也能这样检查。
  - 典型的迭代器
    - 迭代器应该实现 __next__ 和 __iter__ 两个方法，而且这么做能让迭代器通过 issubclass(SentenceIterator, abc.Iterator) 测试。
    - 如果让迭代器类继承abc.Iterator 类，那么它会继承 abc.Iterator.__iter__ 这个具体方法。
  - 构建可迭代的对象和迭代器时经常会出现错误，原因是混淆了二者
    - 可迭代的对象有个 __iter__ 方法，每次都实例化一个新的迭代器；而迭代器要实现 __next__ 方法，返回单个元素，此外还要实现 __iter__ 方法，返回迭代器本身。
    - 因此，迭代器可以迭代，但是可迭代的对象不是迭代器。
    - 常见的反模式
      - 除了 __iter__ 方法之外，你可能还想在 Sentence 类中实现 __next__ 方法，让 Sentence 实例既是可迭代的对象，也是自身的迭代器。
    - 迭代器模式可用来
      - 访问一个聚合对象的内容而无需暴露它的内部表示
      - 支持对聚合对象的多种遍历
      - 为遍历不同的聚合结构提供一个统一的接口（即支持多态迭代）
    - 正确的实现方式是，每次调用 iter(my_ iterable) 都新建一个独立的迭代器。
      - 为了"支持多种遍历"，必须能从同一个可迭代的实例中获取多个独立的迭代器，而且各个迭代器要能维护自身的内部状态
    - 可迭代的对象一定不能是自身的迭代器。
      - 可迭代的对象必须实现 __iter__ 方法，但不能实现 __next__ 方法。
      - 迭代器应该一直可以迭代。迭代器的 __iter__ 方法应该返回自身。
  - 生成器函数
    - return 语句不是必要的：这个函数可以直接"落空"，自动返回。
      - 不管有没有return 语句，生成器函数都不会抛出 StopIteration 异常，而是在生成完全部值之后会直接退出。
    - 生成器函数的工作原理
      - 只要 Python 函数的定义体中有 yield 关键字，该函数就是生成器函数。
        - 调用生成器函数时，会返回一个生成器对象。也就是说，生成器函数是生成器工厂。
        - 把生成器传给 next(...) 函数时，生成器函数会向前，执行函数定义体中的下一个 yield 语句，返回产出的值，并在函数定义体的当前位置暂停。
        - 最终，函数的定义体返回时，外层的生成器对象会抛出 StopIteration 异常——这一点与迭代器协议一致。
      - 定义生成器函数的方式与普通的函数无异，只不过要使用 yield 关键字。
  - 惰性实现
    - 惰性实现是指尽可能延后生成值。这样做能节省内存，而且或许还可以避免做无用的处理。
      - 避免初始化时直接处理整个文本或其他数据，从而节省内存。
    - re.finditer 函数是 re.findall 函数的惰性版本，返回的不是列表，而是一个生成器，按需生成 re.MatchObject 实例。
      - 如果有很多匹配，re.finditer 函数能节省大量内存。
  - 生成器表达式
    - 生成器表达式可以理解为列表推导的惰性版本：不会迫切地构建列表，而是返回一个生成器，按需惰性生成元素。
      - 如果列表推导是制造列表的工厂，那么生成器表达式就是制造生成器的工厂。