Iterators
Browsing on Containers

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Iterators
What is an Iterator?

Iterators are special objects that understand the iterator protocol:

- `__iter__` to build the iterator structure;
- `__next__` to get the next element in the container, and
- `StopIteration` exception to notify when data in container are finished.

Generators are a special case of iterators.

```python
class Fib:
  '''iterator that yields numbers in the Fibonacci sequence'''
  def __init__(self, max):
    self.max = max
  def __iter__(self):
    self.a = 0
    self.b = 1
    return self
  def __next__(self):
    fib = self.a
    if fib > self.max: raise StopIteration
    self.a, self.b = self.b, self.a + self.b
    return fib

if __name__ == "__main__":
   f = Fib(1000)
   for i in f: print(i)
```

Iterators
Lazy Pluralize

```python
class LazyRules:
   def __init__(self, rules_filename):
      self.pattern_file = open(rules_filename, encoding='utf-8')
      self.cache = []
   def __iter__(self):
      self.cache_index = 0
      return self
   def __next__(self):
      self.cache_index += 1
      if len(self.cache) >= self.cache_index:
         return self.cache[self.cache_index - 1]
      if self.pattern_file.closed: raise StopIteration
      line = self.pattern_file.readline()
      if not line:
         self.pattern_file.close()
         raise StopIteration
      pattern, search, replace = line.split(None, 3)
      funcs = build_match_and_apply_functions(pattern, search, replace)
      self.cache.append(funcs)
      return funcs
rules = LazyRules()
```

1. minimal startup cost: just instantiating a class and open a file
2. maximum performance: the file is read on demand and never re-read
3. code and data separation: patterns are stored on a file separated from the code

Iterators
Cryptarithms

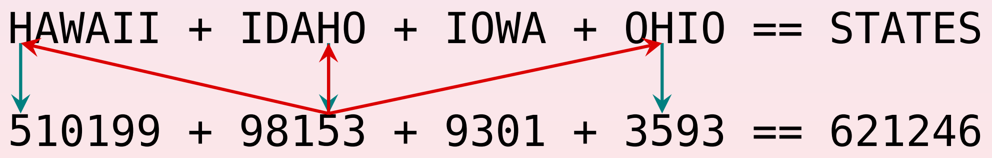
The riddle:

```
HAWAII + IDAHO + IOWA + OHIO == STATES
```

is a cryptarithms

- the letters spell out actual words and a meaningful sentence
- each letter can be translated to a digit (0-9) no initial can be translated to 0
- to the same letter corresponds the same digit along the whole sentence and no digit can be associated to two different letters
- the resulting arithmetic equation represents a valid and correct equation

That is, the riddle above:



```
HAWAII + IDAHO + IOWA + OHIO == STATES

510199 + 98153 + 9301 + 3593 == 621246
```

Iterators
Cryptarithms: the Solution

How can we face the riddle automatic solution?
A brute force approach.

First step consists of organizing the data

- to find the words that need to be translated
- to determine which characters compose such a sentence
- to determine which characters are at the beginning of the words

Then, we look for the solution, if any, by

- generating every possible permutation of ten digits (0-9)
- skimming those permutations with 0 associated to an initial
- trying if the remaining permutations represent a valid solution

Iterators
Cryptarithms: the Solution

```python
import re, itertools, sys

def solve(puzzle):
  words = re.findall('[A-Z]+', puzzle.upper())
  unique_characters = set(''.join(words))
  assert len(unique_characters) <= 10, 'Too many letters'
  first_letters = {word[0] for word in words}
  n = len(first_letters)
  sorted_characters = ''.join(first_letters) + ''.join(unique_characters-first_letters)
  characters = tuple(ord(c) for c in sorted_characters) # generator expression
  digits = tuple(ord(c) for c in '0123456789')
  zero = digits[0]
  for guess in itertools.permutations(digits, len(characters)):
    if zero not in guess[:n]:
      equation = puzzle.translate(dict(zip(characters, guess)))
      if eval(equation): return equation

if __name__ == '__main__':
  for puzzle in sys.argv[1:]:
    print(puzzle)
    solution = solve(puzzle)
    if solution: print(solution)
```

```
[15:06]cazzola@hymir:~/>python3 cryptarithms.py "HAWAII + IDAHO + IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246
```

Iterators

the module itertools: an overview

Combinatoric Generators

– permutations(), combinations(), and so on

```
>>> list(itertools.combinations('ABCD',2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

Infinite Iterators

– count(), cycle() and repeat()

```
>>> list(itertools.repeat('ABCDF',3))
['ABCDF', 'ABCDF', 'ABCDF']
```

Iterators

– zip_longest(), groupby(), islice() and so on

```
>>> list(itertools.starmap(lambda x,y:x+y,itertools.zip_longest('a'*7,'1234567')))
['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7']
>>> names = ['alpha', 'beta', 'gamma', 'delta', 'epsilon', 'zeta', 'eta',
            'theta', 'iota', 'kappa', 'lambda', 'nu', 'mu', 'xi', 'omicron', 'pi',
            'rho', 'sigma', 'tau', 'upsilon', 'phi', 'chi', 'psi', 'omega']
>>> groups = itertools.groupby(sorted(names, key=len), len)
>>> for g, itr in groups: print(list(itr), end=' ')
['nu', 'mu', 'xi', 'pi'] ['eta', 'rho', 'tau', 'phi', 'chi', 'psi']
['beta', 'zeta', 'iota'] ['alpha', 'gamma', 'delta', 'theta', 'kappa', 'sigma',
'omega'] ['lambda'] ['epsilon', 'omicron', 'upsilon']
```

Derived Iterators

```
def enumerate(iterable, start=0):
    return zip(count(start), iterable)

def tabulate(function, start=0):
    """Return function(0), function(1), ..."""
    return map(function, count(start))

def consume(iterator, n):
    """Advance the iterator n-steps ahead.
       If n is none, consume entirely."""
    collections.deque(islice(iterator, n), maxlen=0)

def nth(iterable, n, default=None):
    """Returns the nth item or a default value"""
    return next(islice(iterable, n, None), default)

def quantify(iterable, pred=bool):
    """Count how many times the predicate is true"""
    return sum(map(pred, iterable))

def ncycles(iterable, n):
    """Returns the sequence elements n times"""
    return chain.from_iterable(repeat(iterable, n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    return list(chain.from_iterable(listOfLists))
```

```
def pairwise(iterable):
    """s -> (s0,s1), (s1,s2), (s2, s3), ..."""
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def roundrobin(*iterables):
    # roundrobin('ABC', 'D', 'EF') --> A D E B F C
    # Recipe credited to George Sakkis
    pending = len(iterables)
    nexts =  \
       cycle(iter(it).__next__ for it in iterables)
    while pending:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            pending -= 1
            nexts = cycle(islice(nexts, pending))

def powerset(iterable):
    # powerset([1,2,3]) -->
    # () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return \
      chain.from_iterable(combinations(s, r) \
        for r in range(len(s)+1))
```

Iterators
eval()

eval() is an expression evaluator: it takes a string and evaluates it in the current context.

```
[14:08]cazzola@hymir:~/esercizi-pa>python3
>>> eval('9567 + 1085 == 10652')
True
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> x = 5
>>> eval("x * 5")
25
>>> eval("pow(x, 2)")
25
>>> import math
>>> eval("math.sqrt(x)")
2.23606797749979
>>> def ack(m,n):
...    if m == 0: return n+1
...    elif m>0 and n==0: return ack(m-1,1)
...    else: return ack(m-1, ack(m, n-1))
...
>>> import sys
>>> sys.setrecursionlimit(100000)
>>> eval('ack(2,1000)')
2003
```

```
>>> eval("__import__('subprocess').getoutput('ls -x')")
alphabet-merge.py args.py         counter.py
cryptarithms.py   factorial.py    fib-iterator.py
fibonacci.py      functional      gfib.py
hanoi.py          humanize.py     ifibonacci.py
imp-sieve.py      ls-l.py         matrix.py
modules           oop             plural.py
quicksort.py      sieve.py        sol-eulero.py
sol-fib1000.py    temperatures.py tfact.py
# unsafe! I could evaluate malicious code! solutions?
>>> eval('math.sqrt(x)', {}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError:  name 'math' is not defined
>>> eval('__import__("math").sqrt(x)', {}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError:  name 'x' is not defined
>>> eval('__import__("math").sqrt(x)', {'x': x}, {})
2.23606797749979
# still unsafe! built-ins are available!
>>> eval("__import__('math').sqrt(5)",
...      {"__builtins__":None}, {})
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.
- Mark Summerfield.
  Programming in Python 3: A Complete Introduction to the Python Language.
  Addison-Wesley, October 2009.