COMP10001 Foundations of Computing
CSV files; Iterators and itertools

Semester 1, 2019
Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— version: 1537, date: April 18, 2019 —

Reminders

- Project 2 due Thursday 9/5
- No Grok worksheets due next week!
- Grok worksheets 14 & 15 due Monday 13/5
- Revision lecture tomorrow (Friday)

Lecture Agenda

- Last lecture:
  - Project 1 Review
  - Exception handling
  - Assertions
- This lecture:
  - CSV files
  - Iterators
  - `itertools`

Lecture Outline

Reading files reminder

- You have seen how to read text files using `open()`, `read()` and `readlines()`

```
FILENAME = 'jabberwocky.txt'
text = open(FILENAME).read()
lines = open(FILENAME).readlines()
fp = open(FILENAME)

for line in fp:
    ...
```


CSV files

- A common data format is the CSV ("comma-separated values") files store tabulated data (think spreadsheets) as plain-text, separated by commas:

```
Year,Make,Model
1997,Ford,Falcon
2006,Honda,Odyssey
```

- So common, in fact, that there is a `csv` module that helps you read them

Parsing CSV Files: list of {}

- Use `csv.DictReader` from the `csv` library, which imports the data into a list of dictionaries:

```
import csv
FNAME = 'rainfall.csv'
table = []
for line in csv.DictReader(open(FNAME)):
    table.append(line)
for row in table:
    print(row['city'],row['Dec'])
```


Parsing CSV Files: list of lists

- We can parse CSV files using `csv.reader` from the `csv` library, which imports the data into a list of lists:

```
import csv
FNAME = 'rainfall.csv'
table = []
for line in csv.reader(open(FNAME)):
    table.append(line)
print(table[-1][2])
```

- What if we want to skip the first 3 lines?

Parsing CSV Files: list of lists

```
"""
   Read file into list of lists
   and throw away the first 3 lines.
"""
import csv
FNAME = 'rainfall.csv'
table = []
for line in csv.reader(open(FNAME)):
    table.append(line)

table.pop(0)
table.pop(0)
table.pop(0)
```


Parsing CSV Files: list of lists

```
"""Read file into list of lists
   skipping first 3 lines."""
import csv
FNAME = 'rainfall.csv'
table = []
count = 0
for line in csv.reader(open(FNAME)):
    if count > 2:
        table.append(line)
    count += 1
```

- This looks like a confusion between a very common while loop and a for loop.

For vs While?

What is the difference between `for..in..` and `while`?

- Imagine you could not use `for ... in ...:`
- How would you do this?

```
for element in [1, 2, 3]:
    print(element)
```


For vs While?

What is the difference between `for..in..` and `while`?

- Imagine you could not use `for ... in ...:`
- How would you do this?

```
for element in [1, 2, 3]:
    print(element)
```

- Something like...

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

What do you need to keep track of?

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

- The iterable
- The current index in the iterable
- The end of the iterable

Python has Iterators to do this for you.

Lecture Outline

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

- Definition: an iterable object will return an iterator object when you pass it to the built-in Python function `iter()`.
(This happens automatically with `for..in..:`)

Iterator Objects

- Iterators have a `__next__` method that will return the next thing in the iteration, and update their state/memory of where they are up to. You can access it with the built-in function `next()`
- Iterators raises a `StopIteration` exception when the container is empty

Iterator Objects

- With explicit indexing:

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

- With an iterator:

```
iterator = iter([1,2,3])
while True:
    try:
        print(next(iterator))
    except StopIteration:
        break
```


Why?

- Iterators are more memory efficient than storing and indexing a whole iterable.
  Compare

```
FILENAME = 'jabberwocky.txt'
text = open(FILENAME).readlines()
for line in text:
    ...
```

with

```
FILENAME = 'jabberwocky.txt'
text = open(FILENAME)
for line in text:
    ...
```


Iterators vs. Sequences

Iterators

- no random access
- "remembers" last item seen
- no `len()`
- can be infinite
- traverse exactly once (forwards)

Sequences

- supports random access
- doesn't track last item
- has `len()`
- must be finite
- "traverse" multiple times (fwd/rev/mix)

Why?

- There is less scope for a programmer to make an error
  - You do not have to initialise the index variable
  - You do not have to define the end value of the index variable
  - You do not have to increment the index variable
- Simply create the Iterator using `iter` and step through it using `next`. Python will take care of all the indexing.
- (Similarly, prefer `for` over `while` for a subset of those reasons.)

Iterable vs. Iterator

- *Iterable* describes something that could conceptually be accessed element-by-element
- *Iterator* is an actual interface (or an object implementing the interface) allowing element-by-element access to its contents
- We use an iterator to access an iterable object

Lecture Outline

The itertools Module

- Implements a number of iterator "building blocks"
- Inspired by other programming languages (APL, Haskell, SML)
- Standardises a set of fast, memory efficient tools
- Each tool can be used alone or in combination
- Forms an "iterator algebra"

product: Cross-product of Sequences

```
from itertools import product
def get_deck():
    """Create a list of 52 cards."""
    suits = 'CDHS'
    values = '234567890JQKA'
    deck = product(values, suits)
    return (['',join(c) for c in deck])
```


cycle: Repeating Items Indefinitely

```
from itertools import cycle
def deal():
    """Put cards in 4 equal piles."""
    deck = get_deck()
    hands = [[], [], [], []]
    players = cycle(hands)
    for card in deck:
        player = next(players)
        player.append(card)
    return(hands)
print(deal()[0])
print(deal()[1])
print(deal()[2])
print(deal()[3])
```


Example Uses

```
>>> deck = get_deck()
>>> len(deck)
52
>>> deck[:7]
['3C', '3S', '3D', '3H', '4C', '4S', '4D']
>>> hands = deal()
>>> hand1 = sorted(hands[0])
>>> hand1
['2C', '3H', '3S', '4C', '4S', '5S', '6D',
 '7D', '8D', 'AC', 'AD', 'AH', 'AS']
```


groupby: Group Items by Some Criterion

```
>>> from itertools import groupby
>>> def first(x): return(x[0])
>>> for rank, group in groupby(hand, first):
...   print("{} {}".format(rank, list(group)))
2 ['2C']
3 ['3H', '3S']
4 ['4C', '4S']
5 ['5S']
6 ['6D']
7 ['7D']
8 ['8D']
A ['AC','AD','AH', 'AS']
```


combinations: *n* Choose *k*

```
>>> from itertools import combinations
>>> aces = ['AC','AD','AH', 'AS']
>>> combinations(aces,2)
<itertools.combinations object at 0x1794998>
>>> list(combinations(aces,2))
[('AC', 'AD'), ('AC', 'AH'), ('AC', 'AS'),
 ('AD', 'AH'), ('AD', 'AS'), ('AH', 'AS')]
```

Lecture Summary

- CSV files and the csv library
- What is an iterator?
- Why are iterators useful?
- Differences between sequences and iterators
- The `itertools` module