Introduction to Programming for Public Policy Week 3 (Lists and String Parsing)

Eric Potash

April 10, 2018

Lists

List operations: +

Can concatenate lists just like strings:

```
>>> [1, 2, 3] + ['a', 'b', 'c']
[1, 2, 3, 'a', 'b', 'c']
```

List operations: *

Can also repeat a list with *:

```
>>> [0] * 5
[0, 0, 0, 0, 0]
>>> [1,2,3] * 2
[1, 2, 3, 1, 2, 3]
```

List operator: `in`

You can check whether a value is in a list using the `in` operator:

```
>>> ls = [1,3,5,7]
>>> 3 in ls
True
>>> 4 in ls
False
```

List slices

As with strings we can slice lists. Note that, as with strings, we can omit either end: the slice will then start at the beginning or end at the end of the list.

```
>>> a = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a[2:]
['c', 'd', 'e', 'f']
>>> a[:3]
['a', 'b', 'c']
```

List append

```
>>> a = ['a', 'b', 'c']
>>> a.append('d')
>>> a
['a', 'b', 'c', 'd']
```

List extend

```
>>> a = ['a', 'b', 'c']
>>> a.extend(['d', 'e', 'f'])
>>> a
['a', 'b', 'c', 'd', 'e', 'f']
```

List sort

The `sort` function sorts the list:

```
>>> a = [15, 11, 2, 23, 13]
>>> a.sort()
>>> a
[2, 11, 13, 15, 23]
```

Inplace

Note that the above list functions (`append`, `extend`, `sort`) modify the functions *inplace* and return `None`.

Removing elements

Two ways to remove elements from a list:

- Remove by value:

```
>>> a = ['a','b','c','d']
>>> a.remove('d')
>>> a
['a', 'b', 'c']
```

Removing elements

Two ways to remove elements from a list:

- Remove by value:

```
>>> a = ['a','b','c','d']
>>> a.remove('d')
>>> a
['a', 'b', 'c']
```

- Remove by index and return value:

```
>>> a.pop(1)
'b'
>>> a
['a', 'c']
```

Median

We can use sorting to find the median in a list. Sort and take the middle value:

```
>>> a = [15, 11, 2, 23, 13]
>>> b = sorted(a)
>>> b[round(len(b)/2)-1]
15
```

Percentile

More generally, we can find an arbitrary percentile `p`:

```
>>> b = sorted(a)
>>> b[round(len(b)*p)-1]
```

Note that this is a crude version of percentile– in practice we use interpolation to refine.

String parsing

What is parsing?

The process of reading through a string to break it down or interpret it is called *parsing*.

- Command line programs do this in order to interpret options and arguments
- Python does this to execute your code
- Google does it to execute a search query
- Etc.

String parsing example

```
city_state = 'Chicago, IL'

if ',' not in city_state: # substring
    raise ValueError('no comma')

comma_index = city_state.find(',')
city = city_state[:comma_index]
state = city_state[comma_index+2:]

if len(state) != 2:
    raise ValueError('invalid state abbrev: ' + state)

print('City:', city)
print('State:', state)
```

More string functions

- `string.endswith(ending)`: does `string` end in `ending`?

More string functions

- `string.endswith(ending)`: does `string` end in `ending`?
- `string.startswith(beginning)`: does `string` start with `beginning`?

More string functions

- `string.endswith(ending)`: does `string` end in `ending`?
- `string.startswith(beginning)`: does `string` start with `beginning`?
- `substring in string`: does `string` contain `substring`?

More string functions

- `string.endswith(ending)`: does `string` end in `ending`?
- `string.startswith(beginning)`: does `string` start with `beginning`?
- `substring in string`: does `string` contain `substring`?
- `string.find(substring)`: what is the (first!) index of `substring` in `string`? (or -1 if `substring not in string`)

Command Line Arguments

A python script can use command line arguments through the `argv` list in the `sys` module:

```
# cmd_args.py
import sys
print(sys.argv)
```

```
$ python cmd_args.py -o -h arguments
['cmd_args.py', '-o', '-h', 'arguments']
```

Example: Reading a CSV file

Opening a file

Open a file using the `open` function:

```
>>> file = open('salaries.csv')
>>> print(file)
<_io.TextIOWrapper name='salaries.csv' mode='r' encoding='U
```

This `TextIOWrapper` object facilitates I/O (input/output).

Reading a line

- Read a line from the file:

```
>>> file = open('salaries.csv')
>>> line = file.readline()
>>> line
'Name,Job Titles,Department,Full or Part-Time,
 Salary or Hourly,Typical Hours,Annual Salary,
 Hourly Rate\n'
```

- \n is the *line feed* character. It is a single character. One way to remove it in this example:

```
>>> line[:-1]
'Name,Job Titles,Department,Full or Part-Time,
 Salary or Hourly,Typical Hours,Annual Salary,
 Hourly Rate'
```

Reading many lines

You can iterate over the lines in a file similarly to a list:

```
>>> file = open('salaries.csv')
>>> lines = []
>>> for line in file:
...     lines.append(line[:-1])

['Name,Job Titles,Department,Full or Part-Time,Salary or Ho
 '"AARON,  JEFFERY M",SERGEANT,POLICE,F,Salary,,$101442.00,
 '"AARON,  KARINA ",POLICE OFFICER (ASSIGNED AS DETECTIVE),
...]
```

Splitting fields

This list of lines is not very useful for analysis. The first step is to break up the lines into fields.

```
>>> file = open('salaries.csv')
>>> line = file.readline()[:-1]
>>> line.split(',')
['Name',
 'Job Titles',
 'Department',
 'Full or Part-Time',
 'Salary or Hourly',
 'Typical Hours',
 'Annual Salary',
 'Hourly Rate']
```

2d list

```
>>> file = open('salaries.csv')
>>> lines = []
>>> for line in file:
...     fields = line[:-1].split(',')
...     lines.append(fields)
>>> lines
[['Name',
  'Job Titles',
  'Department',
  'Full or Part-Time',
  'Salary or Hourly',
  'Typical Hours',
  'Annual Salary',
  'Hourly Rate'],
 ['"AARON',
```

2d lists

- `lines` above is a two dimensional (2d) list

2d lists

- `lines` above is a two dimensional (2d) list
  - i.e. it is a list in which each item (a row) is itself a list (of fields)

2d lists

- `lines` above is a two dimensional (2d) list
  - i.e. it is a list in which each item (a row) is itself a list (of fields)
- We can subset rows `lines[2]` or a group of columns with a slice `lines[2:4]`

2d lists

- `lines` above is a two dimensional (2d) list
  - i.e. it is a list in which each item (a row) is itself a list (of fields)
- We can subset rows `lines[2]` or a group of columns with a slice `lines[2:4]`
- `lines[1:]` will return everything except the header

Aggregating

We can aggregate the number of full time employees:

```
count = 0
for line in lines:
    if line[4] == 'F':
        count = count + 1
```

Further string parsing

- The 2d list is an improvement but there's more work to do

Further string parsing

- The 2d list is an improvement but there's more work to do
- Extra characters (double quotes in the names)

Further string parsing

- The 2d list is an improvement but there's more work to do
- Extra characters (double quotes in the names)
- Numbers like salaries are still strings

Further string parsing

- The 2d list is an improvement but there's more work to do
- Extra characters (double quotes in the names)
- Numbers like salaries are still strings
- You'll do this in your assignment this week

Further string parsing

- The 2d list is an improvement but there's more work to do
- Extra characters (double quotes in the names)
- Numbers like salaries are still strings
- You'll do this in your assignment this week
- In the future we'll use existing python modules to parse CSVs

Dictionaries

What is a dictionary?

- Another python data structure is a *dictionary* (called a hashmap in some languages).

What is a dictionary?

- Another python data structure is a *dictionary* (called a hashmap in some languages).
- In a list, indices are integers.

What is a dictionary?

- Another python data structure is a *dictionary* (called a hashmap in some languages).
- In a list, indices are integers.
- In a dictionary, indices can take almost any type.

More on dictionaries

- A dictionary can also be thought of as a mapping between *keys* (indices) and *values*.

More on dictionaries

- A dictionary can also be thought of as a mapping between *keys* (indices) and *values*.
- Each key maps to a value. The keys are unique but the values need not be.

More on dictionaries

- A dictionary can also be thought of as a mapping between *keys* (indices) and *values*.
- Each key maps to a value. The keys are unique but the values need not be.
- The combination of a key and a value is called a *key-value pair* or an *item*.

`dict` construction

- Can construct a list using the `dict` function (so don't call your dictionaries `dict`):

```
>>> d = dict()
>>> d
{}
```

`dict` construction

- Can construct a list using the `dict` function (so don't call your dictionaries `dict`):

```
>>> d = dict()
>>> d
{}
```

- `{}` is an empty dictionary.

`dict` construction

- Can construct a list using the `dict` function (so don't call your dictionaries `dict`):

```
>>> d = dict()
>>> d
{}
```

- `{}` is an empty dictionary.
- Curly braces are the analogue for dicts of the square braces `[]` for lists.

Adding items to a dictionary

Add items to a dictionary using square brackets:

```
>>> eng2esp = dict()
>>> eng2esp['one'] = 'uno'
>>> eng2esp
{'one': 'uno'}
```

Now `eng2esp` maps `'one'` to `'uno'`.

Alternative dictionary constructor

You can also create a `dict` with curly brace syntax:

```
>>> eng2esp = {'one':'uno', 'two':'dos', 'three':'tres'}
```

Dictionaries are unordered

- Dictionaries are *unordered* meaning their items do not have a sequence:

```
>>> eng2esp = {'one':'uno', 'two':'dos', 'three':'tres'
>>> eng2esp
{'one':'uno', 'three':'tres', 'two':'dos'}
```

Dictionaries are unordered

- Dictionaries are *unordered* meaning their items do not have a sequence:

```
>>> eng2esp = {'one':'uno', 'two':'dos', 'three':'tres'
>>> eng2esp
{'one':'uno', 'three':'tres', 'two':'dos'}
```

- So you cannot rely on the order of elements in a dictionary. But it's okay because we index the elements using their keys, not their order:

```
>>> eng2esp['two']
'dos'
```

KeyError

If you try to get a value for a non-existent `key` you'll get a `KeyError`:

```
>>> eng2esp['four']
KeyError: 'four'
```

`in` operator

To determine whether a key is in a dictionary you can use the `in` operator:

```
>>> 'one' in eng2esp
True
>>> 'uno' in eng2esp
False
```

Accessing keys and values

A dictionary has three useful "lists" associated with it:

- `keys()` is like a list of the keys:

```
>>> eng2esp.keys()
dict_keys(['three', 'one', 'two'])
```

Accessing keys and values

A dictionary has three useful "lists" associated with it:

- `keys()` is like a list of the keys:

```
>>> eng2esp.keys()
dict_keys(['three', 'one', 'two'])
```

- `values()` is like a list of the values:

```
>>> eng2esp.values()
dict_values(['tres', 'uno', 'dos'])
```

Accessing keys and values

A dictionary has three useful "lists" associated with it:

- `keys()` is like a list of the keys:

```
>>> eng2esp.keys()
dict_keys(['three', 'one', 'two'])
```

- `values()` is like a list of the values:

```
>>> eng2esp.values()
dict_values(['tres', 'uno', 'dos'])
```

- `items()` is like list of lists, each of which is the pair (length 2) `[key, value]`:

```
>>> eng2esp.items()      # items is like a two dimensior
dict_items([('three', 'tres'), ('one', 'uno'), ('two',
```