Fairfield University
DOLAN
School of Business

Python for Analytics

Lists, Files, and Dictionaries

RSI Chapters 10, 11, and 12

Learning Objectives

- **Theory:** You should be able to explain ...
    - What nesting and accumulation are
    - What aliasing is and why it occurs
    - What it means for data to be persistent
    - What a key-value pair is
- **Skills:** You should know how to ...
    - How to add, replace, and delete data to lists and dictionaries
    - How to make a deep copy of a list or dictionary
    - How to read and write structured data to/from a file

Lists

For managing sequences of items

Lists should be passé by now

- Can store ***sequences*** of values of any type
- List items can have different (mixed) types
- ***Indexed*** by integer locations (starting at `[0]`), with slicing (like `[2:5]`) fully supported

Some new things to try ...

- ***Nesting*** lists inside other lists to simulate matrices (with `[i][j]` indexing)
- Add, remove, replace items as needed
- Concatenate lists with other lists
- Test membership (search for) an item in a list

List Membership & Scanning

We can test for whether a given value is found in a list

```
2 in [1,2,3,4]      # is 2 in the list
   → True
0 in [1,2,3,4]      # is 0 in the list
   → False
```

We can even scan for a value’s first occurrence

```
[1,2,3,4].index(3)  # in what position is 3?
   → 2
```

Nested Sequences

```
bunch = [“Mike”, “Carol”, [“Greg”,“Marcia”,
“Peter”, “Jan”, “Bobby”, “Cindy”]]
bunch[2]          → ['Greg', 'Marcia', ...]
bunch[3]          → IndexError
bunch[2][1]       → ‘Marcia’
bunch[2][1][2]    → ‘r’
bunch[1]          → ‘Carol’
bunch[1][1]       →  ‘a’
bunch[1][1][1]    → IndexError
```

Add, Replace, Delete, Insert

```
area_codes =  [212,646]
area_codes += [347, 718, 917, 929]
area_codes    → [212, 646, 347, 718, 917, 929]
area_codes[2] = 110
area_codes    → [212, 646, 110, 718, 917, 929]
del area_codes[2]
area_codes    → [212, 646, 718, 917, 929]
area_codes[2:2] = [347]
area_codes    → [212, 646, 347, 718, 917, 929]
```

Accumulator Pattern

Problem: Need to mark progress (remember things) as we traverse a list.

Solution: Update an **accumulator** variable each time we access a list item. **Use the += assignment operator.**

```
sum=0
cpy=[]
for v in [2, 5, 2]:
    sum += v            # same as sum = sum + v
    cpy += [v]          # same as cpy = cpy + [v]
```

List Comprehensions

```
x=100
# Use an accumulator to build a list of factors
factors=[]
for i in range(1,x+1):
    if int(x/i)==x/i:
        factors += [i]
```

Not only is the list comprehension fewer lines of code, it also performs faster. The tradeoff is readability, as many programmers don’t know about list comprehensions.

```
# Now do it with a list comprehension
factors=[i for i in range(1,x+1) if int(x/i)==x/i]
```

Append and Concatenate

- Appending `list2` to `list1` modifies `list1`
- Concatenating `list2` to `list1` creates a *new* list

```
list1 = [212,646]
list2 = [347,718,917,929]
list1 + list2 # Concatenating
        → [212,646,347,718,917,929]
list1   → [212,646]
list1 += list2 # Appending
list1   → [212,646,347,718,917,929]
```

Copying vs Aliasing

Immutables make *copies*

```
x=(1,2,3)
y=x
x=(4,5,6)
print(x)
    → (4,5,6)
print(y)
    → (1,2,3)
```

y is a *copy* of the original value of **`x`**.

Mutables make *aliases*

```
x=[1,2,3]
y=x
x=[4,5,6]
print(x)
    → [4,5,6]
print(y)
    → [4,5,6]
```

Use **`y=list(x)`** to make a *copy* instead of an *alias*.

y is really just an *alias* for the variable **`x`**.

List Conversions with `list()`

We can create a copy of any sequence with the list() function.

```
list( (1,2,3))
   → [1,2,3]
list(“abcd”)
   → [‘a’,‘b’,‘c’,‘d’]
list(range(4))
   →  [0,1,2,3]
```

Splitting/Joining Strings

We can split a delimited string into a list of items

```
“a,b,c”.split(‘,’)
   → [‘a’,’b’,’c’]
```

We can also do the reverse, building a delimited string from a list of items.

```
“,”.join([‘a’,’b’,’c’])
   → ‘a,b,c’
```

Files

For making data persistent

Persistent State

Data is ***persistent*** if its state (value) outlives the process that created it.

- If you restart the Python interpreter (or your computer), all data in *memory* is lost but any data in *storage* persists.

Persistent data generally resides within **files**.

- Even if your data is managed by a relational database, it ultimately will reside in one or more files.

Opening and Closing Files

The `open()` function returns a Python object that we can use to read or write data from/to a file. The `close()` method closes it when we are done.

```
f_in=open(<filepath>,“r”) # open for reading
    # read data
f_in.close()
f_out=open(<filepath>,“w”) # open for writing
    # write data
f_out.close()
```

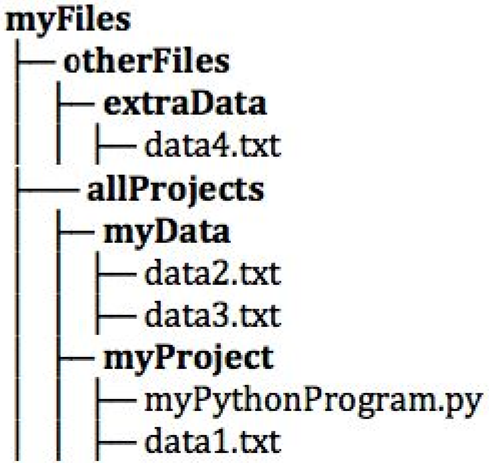
Bug alert! To avoid data corruption, ***never*** open a file for reading and writing at the same.

File Hierarchies and Paths

When opening a file we have to tell Python where to look for it in storage.

Files in storage are logically organized into a hierarchy (tree) of folders and files.

A **file path** gives directions for navigating the tree to the file.

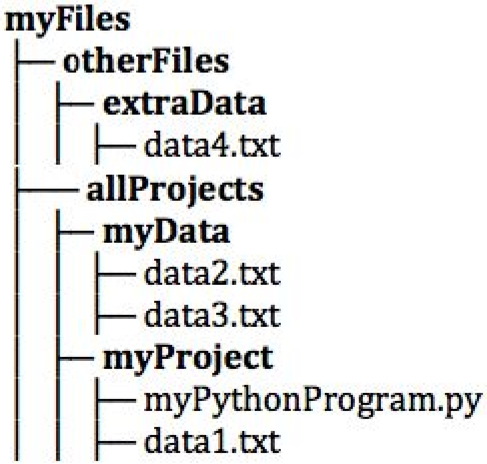


```
myFiles
├─otherFiles
│  ├─extraData
│  │  ├─data4.txt
├──allProjects
│  ├─myData
│  │  ├─data2.txt
│  │  ├─data3.txt
│  ├─myProject
│  │  ├─myPythonProgram.py
│  │  ├─data1.txt
```

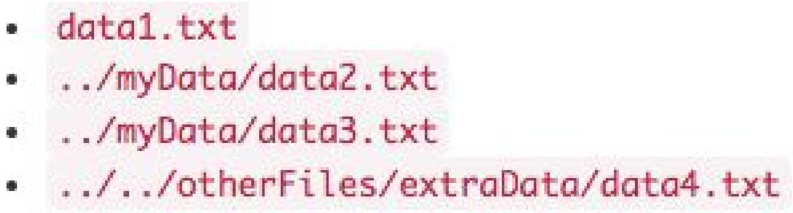
Relative Paths

A **relative path** navigates to the file from where your Python program resides. Each ***segment*** of the path is an instruction:

- Go into a subfolder:
  `<subfolder name>``/`
- Go up a level of the tree:
  `../`
- Open a file in the current folder:
  `<file name>`



```
myFiles
├─otherFiles
│ ├─extraData
│ │ ├─data4.txt
├─allProjects
│ ├─myData
│ │ ├─data2.txt
│ │ ├─data3.txt
│ ├─myProject
│ │ ├─myPythonProgram.py
│ │ ├─data1.txt
```



- `data1.txt`
- `../myData/data2.txt`
- `../myData/data3.txt`
- `../../otherFiles/extraData/data4.txt`

Reading Lines from a Text File

All at once as a list of strings:

```
lines = list(f)
```

Using a `for` loop:

```
for current_line in f:
    # do something with current_line
```

Using `readline()` or `readlines()`:

```
readline()     # read the next line
readlines(n)   # read up to the next n lines
```

Handling Delimited Lines

Each line of the file is a string. Typically, a data file will use a **delimiter** character like a tab, space, or comma to divide each line (string) into **fields**:

```
ID,Waist,Hip,Gender
1,32,40,M
```

We use the string `split()` method to generate a list of strings (one per field) for each line.

```
fields = line.split(‘,’)
```

Writing Text Files

Writing to a text file is similar to reading from one, except that you have to explicitly write the end of line character `\n` yourself.

```
f=open(“myfile.txt”,“w”)
f.write(“a line of text\n”)
f.close()
```

Bug Alert! Always, always, close the file when you are done writing. File systems write data in chunks instead of one character at a time. **Closing the file forces Python to write the last chunk.**

With Statements

To avoid ever forgetting to close a file, use a `with` statement:

```
with open(<filepath>) as <file alias>:
    # do something with the file
with open(“myfile.txt”) as f:
    f.readline() # read the first line
```

Python automatically calls `close()` for us at the end of the `with` statement body.

Dictionaries

For keeping records as (key→ value) pairs

The Mother of All Python Data Types

Dictionaries are ***extremely*** expressive. We can represent almost anything with a dictionary.

- Values can be anything, of course
- Keys can even be any immutable type, including integers

```
listlike_dict = {1:‘a’,2:‘b’,3:‘c’}
listlike_dict[2]
   → ‘b’
```

Add, Replace, Delete (but not Slice)

We can do all of the usual collection operations except those that only work for sequential types.

```
nums = {‘one’:1,‘two’:2,‘three’:3}
nums[‘four’] = 4   # add a key-value pair
nums    →{‘one’:1,‘two’:2,‘three’:3,‘four’:4}
nums[‘four’]= 5    # replace key-value pair
del nums[‘four’]   # delete a key-value pair
nums[‘one’:‘four’] → TypeError
```

Aliasing and Copying Work Too

```
nums = {‘one’:1,‘two’:2,‘three’:3}
ints = nums        # ints is an alias of nums
ints       → {‘one’:1,‘two’:2,‘three’:3}
del nums[‘three’]
ints       →  {‘one’:1,‘two’:2}
ints = dict(nums)  # ints is a copy of nums
nums[‘three’]=3
ints       →{‘one’:1,‘two’:2}
```

Iterating with for Loops

Dictionaries come with a few methods designed to make iteration *really* simple:

```
nums = {‘one’:1,‘two’:2,‘three’:3}
for k in nums.keys():
   print(nums[k])
for v in nums.values():
   print(v)
for (k,v) in nums.items():
   print(k, “maps to”, v)
```

Pro Tip: Use the `zip()` Function

The `zip()` function matches up corresponding items in two equal-length lists to ***generate*** key-value pairs.

```
columns = [‘first_name’, ‘last_name’]
dict( zip(columns, [‘Al’,’Gebra’]) )
   → {‘first_name’:‘Al’,‘last_name’:‘Gebra’}
dict( zip(columns, [‘Betty’,’Boop’]) )
   → {‘first_name’:‘Betty’,‘last_name’:‘Boop’}
```

Fairfield University
DOLAN
School of Business

Python for Analytics

Python Data Types

Python Library Docs