Python L2 - Data Structures, I/O

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

CS302 2019
Hammond Pearce

Data Structures - Lists

- We have already seen the list in action
  - Similar to an array - but maybe more like MATLAB?
- Denoted by square brackets []

```
X = [1, 2, 3, 4]
print(X)
-> [1, 2, 3, 4]
```

Data structures - Lists

- Can have elements of mixed type

```
X = [1,2,3,4]
X.append(“hello”)
print(x)
-> [1, 2, 3, 4, “hello”]
```

Data structures - Lists

- Elements are ordered and accessed by position

```
X = [1,2,3,4, “hello”]
print(x[2])
-> 3
```

- We can access *slices* of the list

```
print(x[2:4])
-> [3,4]
print(x[1:-1])
-> [2,3,4]
```

Data structures - Lists

- Lists are iterable

```
X = [5,2,3,1,4]

for e in X:

    print(“%d, ” % (e + 1), end=‘’)

-> 6, 3, 4, 2, 5,
```

Data structures - Lists

- Lots of useful built-in functions, e.g. sort(), reverse()

```
X = [5,2,3,1,4]

X.sort()

print(X)

-> [1, 2, 3, 4, 5]
```

Data structures - Lists

- There are also similar functions which work on any iterables

```
X = [5,2,3,1,4]
X = sorted(X)
print(X)
-> [1, 2, 3, 4, 5]
```

In-class demo (03.1)

Given a string variable, sort the contents alphabetically

List Comprehension

- List comprehension allows us to be more concise

```
squares = []

for x in range(10):

    square.append(x**2)
```

- is equivalent to

```
squares = [x**2 for x in range(10)]
```

Data structures - Lists

- Elements can be mixed types
- This means elements can also be lists
    - A list of lists, i.e. a matrix

```
matrix = [ [1, 2], [3, 4] ]
```

- Use brackets and commas carefully, and new lines if necessary

Data structures - tuples

- Tuples are similar to lists, except that they are *immutable*
  - Therefore, their values can’t be changed
  - Tuples are also fixed in size
  - They use less memory and run faster, as they are “write-protected”
- Denoted by round parentheses ()
- Can have elements of mixed types, are iterable, can be sliced...

```
tup = (1, 2, 3, “hello”)

print(tup[2])

-> 3

tup.append(4) #error
```

Tuple examples

```
tupA = (1, 2, 3, “hello”)
tupB = (4, ) #extra comma makes this a tuple
tupC = tupA + tupB #tuple concatenation
tupA = tupC
print(tupA)
-> (1, 2, 3, “hello”, 4)
```

- Tuples are often used to return multiple values from a function

```
return (ret1, ret2)
```

Data structure - Sets

- Like lists, but *unordered* and *no duplicates*
  - Used for set mathematics or to remove duplicates from a list
- Denoted using curly braces {}, or ...
- Create a list and then cast to a set

```
x = [1,2,3,4,5,1,2,9,7,3,8,5,4]
x = set(x)
print(x)
-> set([1,2,3,4,5,7,8,9])
```

- Normal sets are iterable and mutable, FrozenSets are immutable

Data Structures - Dictionaries

- Dictionaries are a kind of *key-value* store
  - Lists are a key-value store, where the key is the array position
- Any immutable type can be a key, and any type can be a value
  - Keys must be unique within the dict
  - Access elements like list - []
- Dictionaries are not sortable, and not iterable
  - Although they can be converted to types that are

Dictionary example

```
d = { “name”: “hammond”,
      “age” : 26,
      “job” : “cs302 lecturer” }


print(d[“name”])
-> hammond
print(d[“age”] * 2)
-> 52
```

List Comprehension - Dicts

- List comprehension can be used with dicts too

```
d = {x : x**2 for x in (2,4,6)}

print(d)

-> {2: 4, 4: 16, 6: 36}
```

- There is also a dict constructor

```
d = dict( (2,4), (4,16), (6,36) )

print(d)

-> {2: 4, 4: 16, 6: 36}
```

Data structures - Dictionaries

- If you need to loop over a Dict, loop over its *items*

```
d = {“a” : “b”,
     2   : 4,
     “L” : “OL” }
for key, val in d.items():
    print(“key:”+ str(key) + “, val:”+str(val))


-> key:a, val:b
   key:2, val:4
   key:L, val:OL
```

In-class demo (03.2)

Write a script which contains a dict of people with

{ int(id): { “name” : str(name), “age”: int(age)}}

Write code which can:

1. *Print the data out presentably*
2. *Sort the data before printing it, in order of ID number*
3. *Sort the data before printing it, in order of student name*
4. *Sort the data before printing it, in order of student age*

Terminal I/O

- `input()` and `raw_input()` are the normal input methods
- `print()` is the standard output method
- Both methods are blocking
- Python has a lot of ways to edit the strings passed to print
    - Using %, .format()

```
x = input()

print(x)
```

File I/O

- Python has file I/O built in
- Create file objects by opening them
  - Use the correct flag
- Remember to close them afterwards!

```
in_file = open(“list.txt”, “r”)
data = in_file.read()
in_file.close()
```

File I/O

- “r” is for read-only (text files)
- “rb” is for read-only (binary files)
- “w” is for write-only, overwrites the file
- “a” is for appending, writes from the bottom of file
- “r+” is for reading and writing
- “a+” is for reading and appending

File I/O

- There are plenty of built-in functions for dealing with files
  - read(), readlines(), write()
  - Check the documentation!
- Be careful with endline characters!
  - In Windows, “\r\n”
  - In Linux, “\n”
- Remember to close your files!
- Use the “os” module to move files, check dirs, etc

In-class demo (03.3)

- *Extend the previous demo by getting it to load the students from a file*
- *The data should be saved in the form “id,name,age”*
  - *(the str.split() function is helpful)*
- *Then, it should be loaded into the dict used earlier*

Then, get the program to ask via a prompt which sorting method to use

File I/O

- Common file formats:
  - CSV
  - JSON
  - XML
  - pickle
- Helper modules exist for all of these
- Sometimes, though, it’s best to just use a database…
  - (We’ll cover these later).

Conclusions

- Lists, Sets, Tuples, and Dicts are good ways to store “arrays” of data
  - Varying features for each
  - These provide good ways to manage data within applications
- Program input and output can come via
  - terminal
  - files
  - etc
- Several existing methods for defined storage of data
  - JSON
  - XML
  - etc