Lecture 21

Chapter 12 *More Python Containers*

12.1 Two Familiar Containers: `list` and `tuple`

12.2 Dictionaries

12.3 Containers of containers

12.4 Set and Frozenset

12.5 Arrays

Chapter 12 More Python Containers

Certain classes are designed to provide storage and management of a large collection of objects.
We call them containers.

By design, Python's containers all support certain common syntax, yet they are different in their menu of behaviors and in the underlying efficiencies of their operations.

Let's discuss the following aspects of a container:
order
mutability
associativity
heterogeneity
storage

Introduction

order:
`list`, `tuple`, and `array` are used to represent *ordered sequence* of elements.
(each of these use concept of an integer *index*)

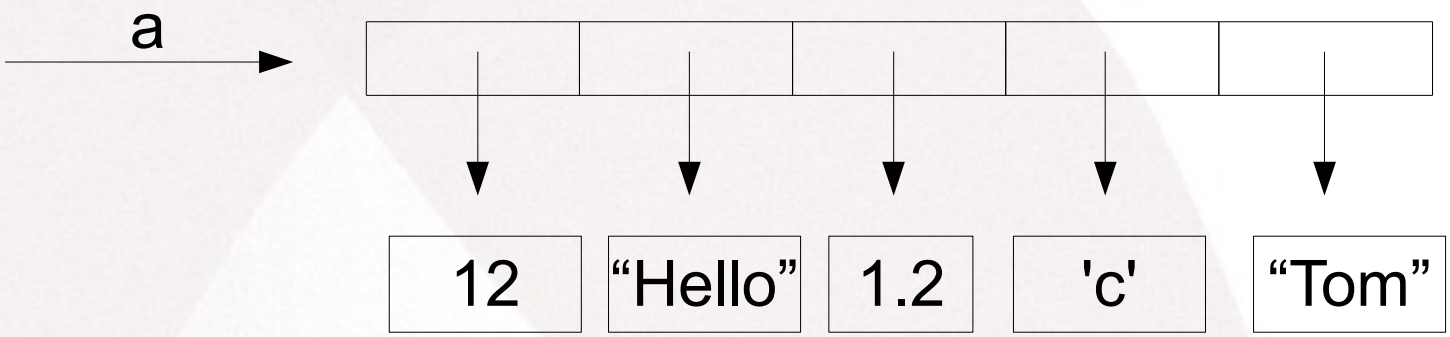
mutability:
`list` is mutable (tuple is not) – we can insert, replace, or remove elements.

associativity:
sometimes we need to associate an element with some data : `dict` class (a.k.a., “dictionary”) can be used for representing an association between a **key** and its **underlying value**.

heterogeneity:
when elements of a container are of different types;
most Python's containers support heterogeneity
(**homogeneous** – all elements are of the same type)

Introduction

storage:
for high-performance operations we don't want to have references to the values of the elements in a container (as in `list` class), but to store the actual values of elements within the state of the container;



`array` class provides compact storage for a collection of data drawn from a chosen primitive type.

Introduction

	list	tuple	dict	set	frozenset	array
ordered	v	v				v
mutable	v		v	v		v
associativity			v			
heterogeneity	v	v	v	v	v	
storage						v

summary table

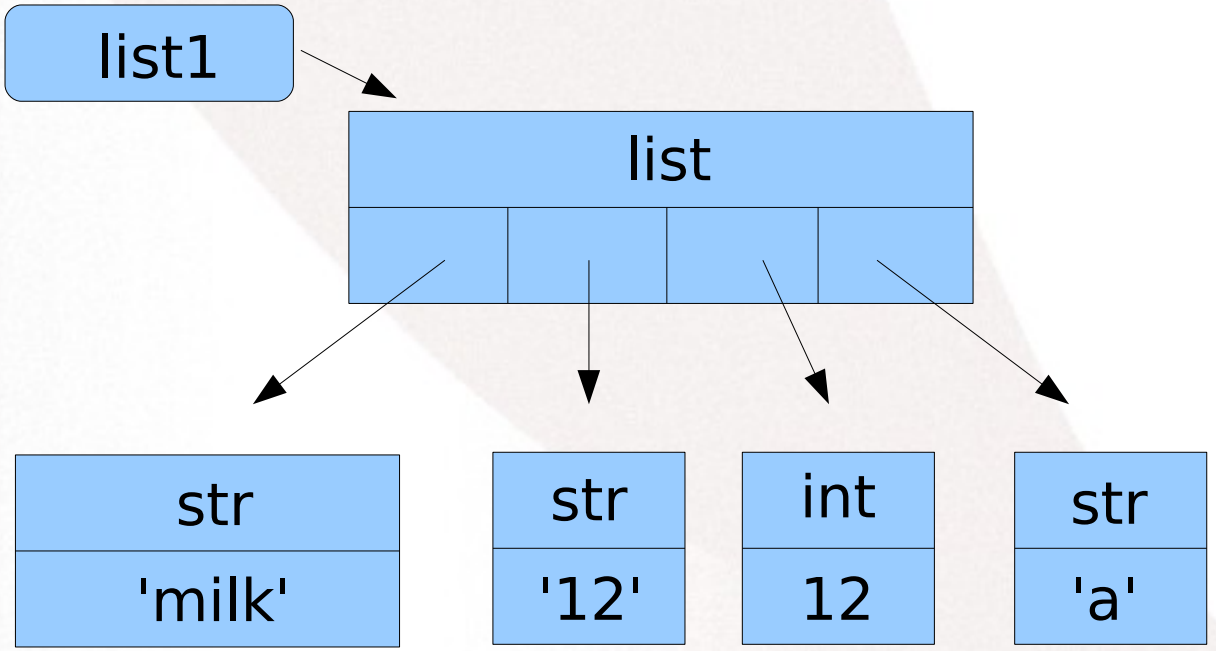
12.1 Two Familiar Containers: `list` and `tuple`

These two containers are used to manage *ordered sequence* of elements.
The position of a particular element within that sequence is designated with an *index*. Indices start with 0.

Examples:

```
list1=['milk','12',123,'a']
tuple1=('milk','apple','123',12)

list[1] → '12'
tuple[3] → 12
...
```



12.1 Two Familiar Containers: `list` and `tuple`

Limitations to the use of a list and a tuple:

Assume that a company assigns each employee an "employee ID" that is an integer identifier. If the company assigns consecutive IDs (perhaps sequentially as they are hired), those IDs can be used as indices into a list.

In this way, an expression such as `employee[235]` might be used to access a particular employee's record.

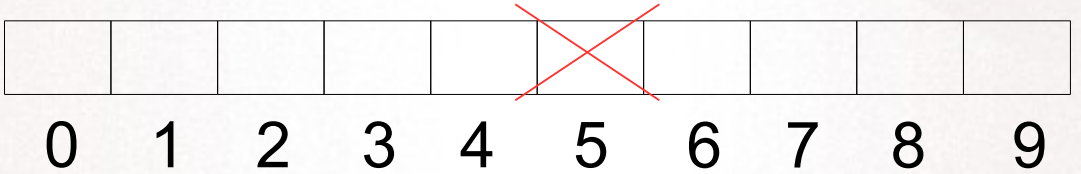
12.1 Two Familiar Containers: `list` and `tuple`

Limitations to the use of a list and a tuple:

Assume that a company assigns each employee an "employee ID" that is an integer identifier. If the company assigns consecutive IDs (perhaps sequentially as they are hired), those IDs can be used as indices into a list.

In this way, an expression such as `employee[235]` might be used to access a particular employee's record.

Problem 1: when employee leaves the company
If `employee[5]` decides to retire and is removed from the list, this causes many other employees to be repositioned within the list.



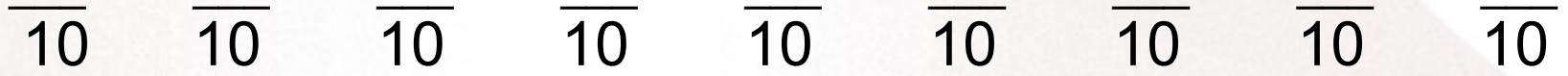
12.1 Two Familiar Containers: `list` and `tuple`

Limitations to the use of a list and a tuple:

Assume that a company assigns each employee an "employee ID" that is an integer identifier. If the company assigns consecutive IDs (perhaps sequentially as they are hired), those IDs can be used as indices into a list.

In this way, an expression such as `employee[235]` might be used to access a particular employee's record.

Problem 2: if employees' identifiers are not consecutively numbered by nature
(some of the companies use SSN to identify employees – if they are used as indices of a list – the length of a list is 1 billion)



12.1 Two Familiar Containers: `list` and `tuple`

Limitations to the use of a list and a tuple:

Assume that a company assigns each employee an "employee ID" that is an integer identifier. If the company assigns consecutive IDs (perhaps sequentially as they are hired), those IDs can be used as indices into a list.

In this way, an expression such as `employee[235]` might be used to access a particular employee's record.

Problem 3: *non-numeric identification*
If we wish to keep a track of our favorite movies and the directors of those movies.
A possible ID: title of the movie
Therefore, director['Star Wars'] would give us the director
- *but it cannot be done with list or tupple*

12.1 Two Familiar Containers: `list` and `tuple`

Limitations to the use of a list and a tuple:

Another example: list of world capitals

There is no natural numbering of countries, therefore capital['Bolivia'] looks very natural, *but also is not supported by lists*.

12.2 Dictionaries

A dictionary represents a mapping from objects known as ***keys*** to associated objects known as ***values***.

Keys can be consecutive integers, or can be drawn from more general domains.

Example: we can use dictionaries to represent mapping from movies to directors

```
director['Star Wars'] → 'George Lucas'
director['The Godfather'] → 'Francis Ford Coppola'
director['American Graffiti'] → 'George Lucas'
```

Title is the *key*, the name of the associated director is the *value*.

The elements of dictionaries are not inherently ordered.

12.2 Dictionaries

Keys

- serve as identifiers
  (when accessing a particular value in collection),
- do not represent a position within the collection,
- do not need to be consecutive integers,
- do not need to be integers,
- required to be unique,
- a tuple can be used for a key
- all keys should be drawn from immutable class (int, str, or tuple)

Python uses the *key* when deciding where to store an entry, and also uses the *key* when later trying to access that entry.

Values

No restrictions on allowable values in a dictionary

12.2 Dictionaries

Syntax:

```
# create an empty dictionary, using constructor of dict class
director = dict()

# add elements
director['Star Wars'] = 'George Lucas'
director['The Godfather'] = 'Francis Ford Coppola'
director['American Graffiti'] = 'George Lucas'
```

Or

```
director = { 'Star Wars':'George Lucas',
'The Godfather' : 'Francis Ford Coppola',
'American Graffiti' : 'George Lucas'}
```

See Table of page 404 for the list of supported behaviors

12.2 Dictionaries

Examples:

```
keys()
# get the list of movie titles, sort them and display them
titles = director.keys()
print(sorted(titles))

# print all keys elements of the dictionary 'director'
for entry in director:
   print(entry)

values()
# print the sorted list of directors
for person in sorted(director.values()):
   print(person,'is a director')

items()
#  iterate over (key,value) pairs and display them
for movie,person in director.items():
   print(movie,'was directed by',person)
```

12.2 Dictionaries

The objects returned by dict.keys(), dict.values() and dict.items() are ***view objects***.

They provide a *dynamic view* on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

12.4 Set and Frozenset

- **set** and **frozenset** are used to represent the classical mathematical notion of a set, an *unordered* collection of *unique* elements
- **set** is mutable, **frozenset** is immutable
- elements added must be immutable
- implemented similar to a dictionary, but without storing associated values
- check for presence in the set/frozenset is very fast
- can be also used for removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference

Example: a set of colors

```
colors = Set(["blue","red","yellow","black","white"])
```

Syntax (with alternate)	Semantics
len(s)	Returns the cardinality of set s.
v **in** s	Returns **True** if set s contains the value v, **False** otherwise.
v **not in** s	Returns **True** if set s does not contain the value v, **False** otherwise.
for v **in** s:	Iterates over all values in set s (in arbitrary order).
s == t	Returns **True** if set s and set t have identical contents, **False** otherwise (order is irrelevant).
s < t	Returns **True** if set s is a *proper* subset of t, **False** otherwise.
s <= t s.issubset(t)	Returns **True** if set s is a subset of t, **False** otherwise.
s > t	Returns **True** if set s is a *proper* superset of t, **False** otherwise.
s >= t s.issuperset(t)	Returns **True** if set s is a superset of t, **False** otherwise.
s | t s.union(t)	Returns a new set of all elements that are in either set s or set t (or both).
s & t s.intersection(t)	Returns a new set of all elements that are in both set s and set t.
s – t s.difference(t)	Returns a new set of all elements that are in set s but not in set t.
s ^ t s.symmetric_difference(t)	Returns a new set of all elements that are in either set s or set t but not both.

p. 411 in textbook

FIGURE 12.4: Accessor methods supported by the **set** and **frozenset**.

Syntax (with alternate)	Semantics
s.add(v)	Adds value v to the set s; has no effect if already present.
s.discard(v)	Removes value v from the set s if present; has no effect otherwise.
s.remove(v)	Removes value v from the set s if present; raises a KeyError otherwise.
s.pop( )	Removes and returns arbitrary value from set s.
s.clear( )	Removes all entries from the set s.
s |= t s.update(t)	Alters set s, *adding* all elements from set t; thus set s becomes the union of the original two.
s &= t s.intersection_update(t)	Alters set s, *removing* elements that are not in set t; set s becomes the intersection of the original two.
s −= t s.difference_update(t)	Alters set s, *removing* elements that are found in set t; set s becomes the difference of the original two.
s ^= t s.symmetric_difference_update(t)	Alters set s to include only elements originally in exactly one of the two sets, but not both.

FIGURE 12.5: Behaviors that mutate a **set**.

p. 413 in the textbook

12.4 Set and Frozenset

Why do we need frozenset?
- elements of set must be immutable;
frozenset is immutable, so we can create a “set of frozensets”

12.3 Containers of containers

The elements of a `list`, `tuple` and the values of a `dict` class can be whatever type of object we want.

So it is perfectly acceptable to maintain a list of lists, a tuple of dictionaries, a dictionary of lists, and so on.

Example: Tic-tac-toe game

	0	1	2
0	X	O	X
1	X	X	O
2	O	X	O

columns

rows

`board=[[X,O,X],[X,X,O],[O,X,O]]` – row by row
row-major order

or

`board =[[X,X,O],[O,X,X],[X,O,O]]` – column by column
column-major order

12.3 Containers of containers

The elements of a `list`, `tuple` and the values of a `dict` class can be whatever type of object we want.

So it is perfectly acceptable to maintain a list of lists, a tuple of dictionaries, a dictionary of lists, and so on.

Example: Tic-tac-toe game

	0	1	2
0	X	O	X
1	X	X	O
2	O	X	O

columns

rows

`board=[[X,O,X],[X,X,O],[O,X,O]]` – row by row
`board[1,2]` – 1st row, 2nd column

or

`board =[[X,X,O],[O,X,X],[X,O,O]]` – column by column
`board[2,1]` - 2nd column, 1st row

12.3 Containers of containers

The elements of a `list`, `tuple` and the values of a `dict` class can be whatever type of object we want.

So it is perfectly acceptable to maintain a list of lists, a tuple of dictionaries, a dictionary of lists, and so on.

Example: Tic-tac-toe game

	0	1	2
0	X	O	X
1	X	X	O
2	O	X	O

columns

rows

like working with arrays

`board=[[X,O,X],[X,X,O],[O,X,O]]` – row by row
`board[1,2]` – 1st row, 2nd column

or

`board =[[X,X,O],[O,X,X],[X,O,O]]` – column by column
`board[2,1]` - 2nd column, 1st row

12.5 Arrays

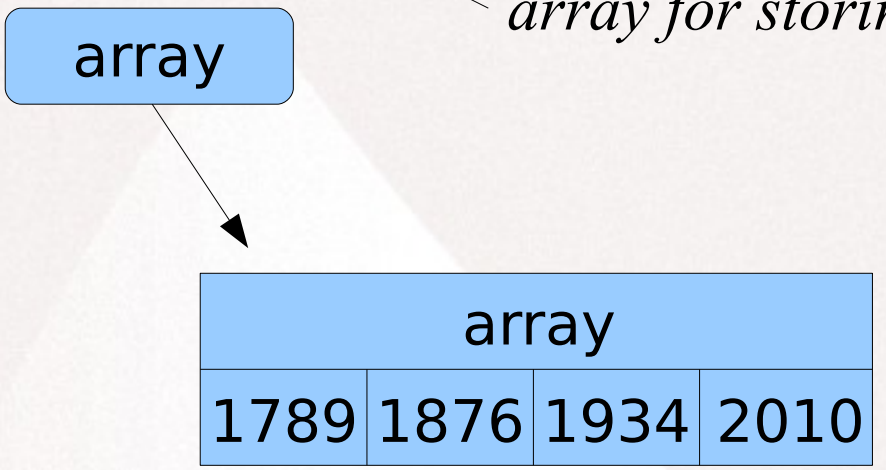
The array class is not among the built-in types of Python.

```
from array import *
```

Let's create an array of integers:

```
yearArray=array('i',[1789,1876,1934,2010])
```

array for storing integers



See program arrays-example.py

Two-dimensional data

Assume that we have the following data:

```
 12  23   34   45
 29  34  123   56
127  98   76  143
```

3 rows, 4 columns

integer values separated by a space

Assume that these values are stored in a file.

Let's write a program that reads all of these values and stores them somewhere.

Two-dimensional data

```
 12  23   34   45
 29  34  123   56
127  98   76  143
```

Here is a sketch of the program:

```
def main():
    # open file
    fname = input('input the name of the file:')
    f = open(fname)
    # store all the data in a list
    data = f.readlines()
    # form a two-dimensional array
    A = processData(data)
    # display the information
    display(A)

main()
```

Two-dimensional data

```
 12  23  34  45
 29  34 123  56
127  98  76 143
```

```
 12  23  34  45

 29  34 123  56

127  98  76 143
```

```
data = ['12 23 34 45',' 29 34 123 56','127 98 76 143']
```

Two-dimensional data

```
 12  23  34  45
 29  34 123  56
127  98  76 143
```

```
 12  23  34  45

 29  34 123  56

127  98  76 143
```

```
data = ['12 23 34 45',' 29 34 123 56','127 98 76 143']
```

```
 12          23          34          45

 29          34         123          56

127          98          76         143
```

```
data = [[12,23,34,45],[29,34,123,56],[127,98,76,143]]
```

HW assignment

implement addition of two matrices

- each input matrix is stored in a file
- the new matrix should be displayed and save in a output file