CS177 Python Programming

Recitation 11 – Data Collections

Table of Contents

- Review the use of lists (arrays) to represent a collection of related data.
- Review the functions and methods available for manipulating Python lists.
- Review the use of other data collections in Python, such as dictionaries and tuples.

Sequences

- When we have multiple ***elements*** stored consecutively in memory we call it a **List**
- When we have multiple ***characters*** stored consecutively in memory we call it a **String**
- Both of these structures are sequence structured (individual items can be selected by indexing i.e. `s[i]`).
- **Ranges**, **Lists** and **Strings** are **0** indexed

Lists

- Lists are defined by **[]**
- Lists can contain *strings*, *numbers*, even other *lists*.

Print a string
(A string of characters) →

Print a list
(A list of characters) →

```
>>>x = “ABCD”
>>>y = [“A”,”B”,”C”,”D”]
>>>print (x)
ABCD
>>>print (y)
['A', 'B', 'C', 'D']
```

Lists

- Lists are more “general purpose”
  - Allow *heterogeneous* elements in the same list

```
>>> myList = ["X", "B", 3, "A", 1]
>>> print (myList)
['X', 'B', 3, 'A', 1]
>>> myList = [['X', 'B', 3, 'A', 1], ‘hello’, 99]
```

[] Notation in Lists

- a[i] : gives a name to the ***i*th** element of a sequence
- The **[]** can be used to *index* into lists, ranges, or strings.
- If the sequence is a *list*, e.g., a = list(range(0, 10))
  - a[i] is equal to the **i+1** number in the range of 0 to 9 (index *starts* from **0**)

Lists Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

Slicing

```
>>>x = list(range(0,10))
```

Print the first element

```
>>>print (x[0])
0                    Get element!
```

Print the first five elements

```
>>>print (x[0:5])
[0,1,2,3,4]          Get sublist!
```

Print the first three elements
Starting point is omitted, default value is 0

```
>>>print (x[:3])
[0,1,2]              Get sublist!
```

Print from the fourth element until the end
Ending point is omitted

```
>>>print (x[3:])
[3,4,5,6,7,8,9]      Get sublist!
```

Print from the last element until the end
Ending point is omitted

```
>>>print (x[-1:])
[9]                  Get sublist!
```

Print from the first element to the last
Starting point is omitted, default value is 0

```
>>>print (x[:-1])
[0,1,2,3,4,5,6,7,8]  Get sublist!
```

Examples of List Operations

```
>>> print([1,2] + [3,4])

>>> print([1,2]*3)

>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> print(grades[0])

>>> print(grades[2:4])

>>> print(len(grades))
```

Examples of List Operations

```
>>> print([1,2] + [3,4])
[1, 2, 3, 4]
>>> print([1,2]*3)
[1, 2, 1, 2, 1, 2]
>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> print(grades[0])
'A'
>>> print(grades[2:4])
['C', 'D']
>>> print(len(grades))
5
```

Lists Operations (con't)

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

Examples of List Operations

```
>>> a=[]
>>> for i in range(15, 3, -2):
        a.append(i)
>>> print(a)

>>> print(a.reverse())

>>> print(a.index(7))
```

Examples of List Operations

```
>>> a=[]
>>> for i in range(15, 3, -2):
        a.append(i)
>>> print(a)
[15, 13, 11, 9, 7, 5]
>>> print(a.reverse())
[5, 7, 9, 11, 13, 15]
>>> print(a.index(7))
1
```

Examples of List Operations

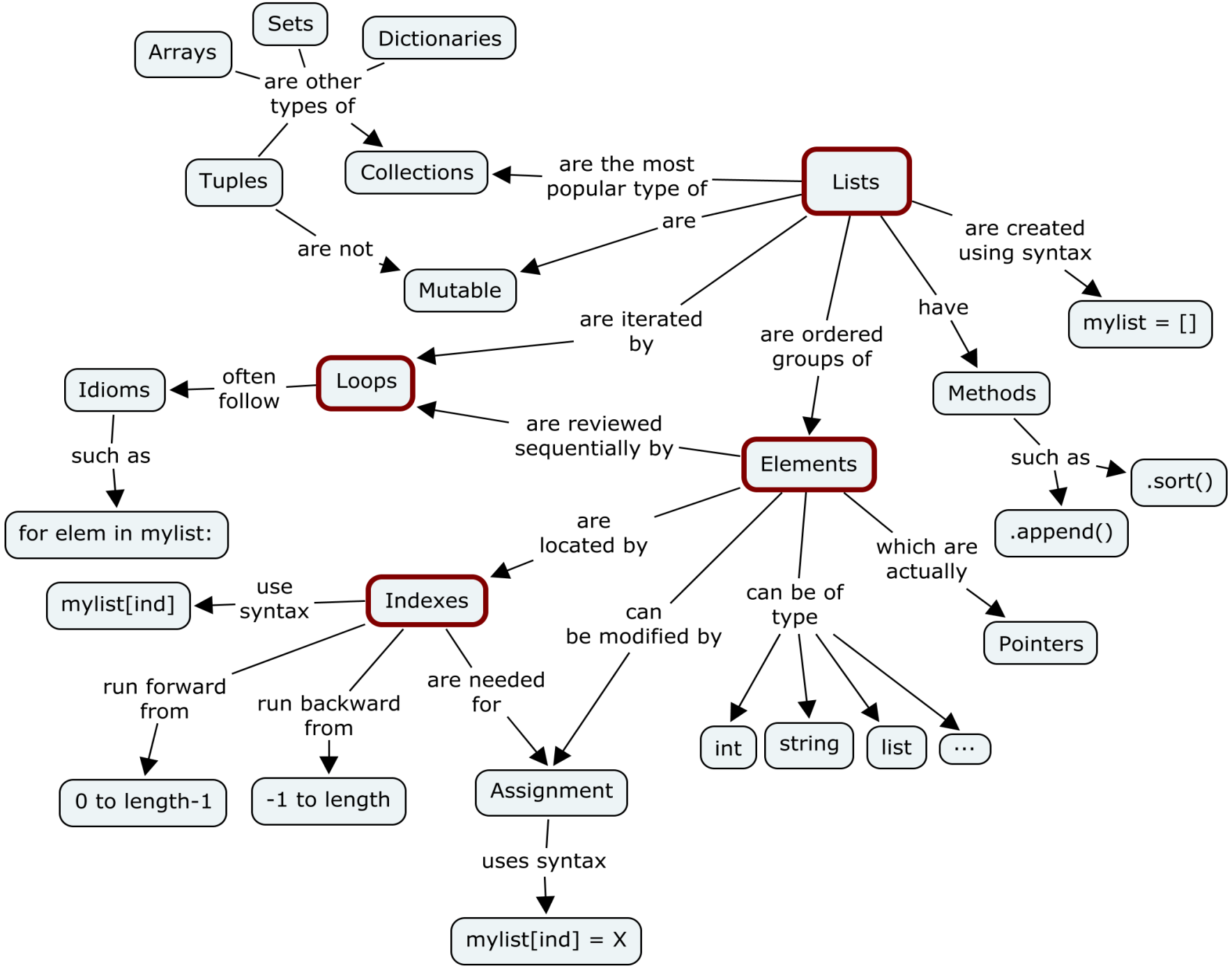
```
>>> a.insert(2, 15)
>>> print(a)

>>> print(a.count(15))

>>> a.remove(15)
>>> print(a)

>>> print(a.pop(2))
```

Examples of List Operations

```
>>> a.insert(2, 15)
>>> print(a)
[5, 7, 15, 9, 11, 13, 15]
>>> print(a.count(15))
2
>>> a.remove(15)
>>> print(a)
[5, 7, 9, 11, 13, 15]
>>> print(a.pop(2))
9
```

Dictionaries, Sets, Tuples

- A collection of **unordered** values accessed by key rather than by index is called **Dictionary**
- A collection of **unordered** and non-duplicated elements is called **Sets**
- In Python a **Tuple** is much like a list except that it is immutable (unchangeable) once created, i.e. (42, 56, 7)
- Note: Being an unordered collection, dic/sets do not record element position or order of insertion. Accordingly, indexing, slicing, or other sequence-like behavior are not supported.


Sets
Arrays
Dictionaries
are other types of
Tuples
Collections
are the most popular type of
Lists
are
are not
Mutable
are created using syntax
mylist = []
are iterated by
are ordered groups of
have
Idioms
often follow
Loops
are reviewed sequentially by
Elements
Methods
such as
.sort()
.append()
such as
for elem in mylist:
are located by
which are actually
Pointers
mylist[ind]
use syntax
Indexes
can be modified by
can be of type
run forward from
run backward from
are needed for
int
string
list
...
0 to length-1
-1 to length
Assignment
uses syntax
mylist[ind] = X

Strings, Lists, Dictionaries, Sets, Tuples

Data Type	Description
List	Sequential, ordered, can have duplicates, mutable, i.e. [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]
String	Sequential, ordered, can have duplicates, immutable, i.e. “hello”
Dictionary	Non-sequential, non-ordered, unique keys, can have duplicates, mutable, i.e. {1:‘h’, 2:‘e’, 3:‘l’, 4:‘l’, 5:‘o’}
Set	Non-sequential, non-ordered, non-duplicate, mutable, i.e. setset(('h','e', 'l', 'o'))
Tuple	Sequential, ordered, can have duplicates, immutable, i.e. tuple(('h','e', 'l', 'l', 'o'))

Dictionary Methods

- A dictionary is an unordered set of *key: value* pairs
- len(d)

Return the number of items in the dictionary *d*.

- d[key]

Return the item of *d* with key *key*.

Raises a KeyError if *key* is not in the map.

- d[key] = value

Set d[key] to *value*.

Dictionaries Methods

- del d[key]

Remove d[key] from *d*.

Raises a KeyError if *key* is not in the map.

- key in d

Return True if *d* has a key *key*, else False.

Example

```
>>> d = {}   #empty dictionary

>>> d = {‘date’ : 18}
#set ‘date’ maps to 18

>>> d[‘date’] = 20
#change the value mapped to by the key ‘date’ to 20
```

Example

Is this right??

If yes, what is the output?

If no, why?

```
>>> d = {'alice' : 1, 'bob' : 2, 'calie': 1}
>>> d =  {'alice' : 1, 'bob' : 2, 'alice': 3}
```

Example

Given a dictionary *dic* and a list *lst*, remove all elements from the dictionary whose key is an element of *lst*. For example, given the dictionary {1:2, 3:4, 5:6, 7:8} and the list [1, 7], the resulting dictionary would be {3:4, 5:6}. Assume every element of the list is a key in the dictionary.

Example

Given a dictionary *dic* and a list *lst*, remove all elements from the dictionary whose key is an element of *lst*. For example, given the dictionary {1:2, 3:4, 5:6, 7:8} and the list [1, 7], the resulting dictionary would be {3:4, 5:6}. Assume every element of the list is a key in the dictionary.

```
for e in lst :
    del dic[e]
```

Sets Methods

- S.update(*t*)

Return set S with element added from t

- S.add(x)

Add element x to set S

- S.remove(x)

Remove x from set S, raises *KeyError* if not present

Example

```
>>> engineers = set(['John', 'Jane', 'Jack',
'Janice'])
>>> engineers.add('Marvin')

>>> employees = set()
>>> employees.update(engineers)

>>> employees.remove('Jack')
```

Example

```
>>> engineers = set(['John', 'Jane', 'Jack', 'Janice'])
>>> engineers.add('Marvin')
{'Jack', 'Marvin', 'Janice', 'John', 'Jane'}
>>> employees = set()
>>> employees.update(engineers)
{'John', 'Jane', 'Janice', 'Jack', 'Marvin'}
>>> employees.remove('Jack')
{'John', 'Jane', 'Janice', 'Marvin‘}
```

Examples

Given the string *line* , create a set of all the vowels in *line*. Associate the set with the variable *vowels*.

Examples

Given the string *line* , create a set of all the vowels in *line*. Associate the set with the variable *vowels*.

```
vowels = set()
for x in line:
    if x=="a" or x=="e" or x=="i" or x=="o" or x=="u":
        vowels.add(x)
```

Tuples

```
>>> t = (12345, 54321, 'hello!' )
>>> print (t)
(12345, 54321, 'hello!')
>>> # Tuples may be nested: ...
>>> u = (t, (1, 2, 3, 4, 5))
>>> print(u)
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Examples

```
>>> # Tuples are immutable: ...
>>> t[0] = 88888
Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects: ... v = ([1, 2, 3], [3, 2, 1])
>>> v[1][2] = 99
>>>print(v)
```

([1, 2, 3], [3, 2, 99])

Examples

Given that *t* has been defined and refers to a tuple write some statements that associate with *t* a new tuple containing the same elements as the original but in sorted order.

Examples

Given that *t* has been defined and refers to a tuple write some statements that associate with *t* a new tuple containing the same elements as the original but in sorted order.

```
tmp = list(t)
tmp.sort()
t = tuple(tmp)
```