MAT 305: Mathematical Computing

Collections

John Perry

University of Southern Mississippi

Fall 2011

Outline

1. Collections in Python
2. Ranges of data
3. Strings
4. Summary

Collections?

Collection: group of objects identified as single object

- ordered
  - points (x_0, y_0), (x_0, y_0, z_0)
  - tuples $(a_0, a_1, a_2, \ldots a_n)$
  - lists $[a_0, a_1, \ldots, a_n]$
  - sequences $(a_0, a_1, a_2, \ldots)$
- unordered
  - sets $\{a_0, a_5, a_3, a_2, a_1\}$

Outline

1. Collections in Python
2. Ranges of data
3. Strings
4. Summary

Python collections

Sage offers several collections standard in Python

- ordered ("sequence types")
  - tuples, lists
  - access ith element using `[i-1]`
- unordered ("set types")
  - sets
  - cannot access ith element
  - only one instance of any element

Tuples

A **tuple** is an immutable, ordered collection

- *immutable*: cannot change elementsn
- *indexable*: can access elements by their order
- defined using parentheses

Example

```
sage:  my_tuple = (1,5,0,5)                    4-tuple

sage:  my_tuple[2]                             access 3rd element
0

sage:  my_tuple[2] = 1                         assign to 3rd element?
...Output deleted...
TypeError:  'tuple' object does not support item
assignment

sage:  my_tuple
(1,5,0,5)
```

Lists

A **list** is a mutable, ordered collection

- *mutable*: can change elements
- *indexable*: can access elements by their order
- defined using square brackets

Example

```
sage:  my_list = [1,5,0,5]
```
list of 4 elements

```
sage:  my_list[2]
0
```
access 3rd element

```
sage:  my_list[2] = 1
```
assign to 3rd element?

```
sage:  my_list[2]
1
```
no error! access gives new value!

```
sage:  my_list
[1,5,1,5]
```

Sets

A **set** is a mutable, unordered collection

- *mutable*: can change elements
- *non-indexable*
  - cannot access elements by their order
  - computer arranges elements for efficiency
- defined using `set(`*tuple or list*`)` or `set()` (for empty set)
- redundant elements automatically deleted

Example

```
sage: my_set = set([1,5,0,5])
```
set of 4 elements

```
sage: my_set[2]
```
access 3rd element?

...Output deleted...

```
TypeError: 'set' object is unindexable
```

```
sage: my_set
```
so what's in there, anyway?

```
set([0, 1, 5])
```
not original list!

Nice dog! Does he do any tricks? (1)

sets, tuples, and lists

- `len(`*C*`)`
  number of elements in C
- *x* `in` *C*
  is x *an element of* C*?*

tuples and lists

- *C*`.count(`*x*`)`
  Number of times x *appears in* C
- *C*`.index(`*x*`)`
  First location of x *in* C
- *C1* + *C2*
  join *C1* to *C2*, returned as new tuple/list

Example

```
sage: len(my_set)
3

sage: 4 in my_set
False

sage: 5 in my_set
True

sage: my_tuple.count(5)
2

sage: my_list.index(5)
1

sage: my_list + [1,3,5]
[1, 5, 0, 5, 1, 3, 5]
```

How many 5s?

in second location

Nice dog! Does he do any tricks? (2)

lists

- L`.append(`x`)`
- L`.extend(`C`)` — *append each element of* C to L
- L`.insert(`i`, `x`)` — *insert* x *at* L`[`i`]`, *shifting* $L[i]$ *and subsequent elements back*
- L`.pop(`i`)` — *delete* $L[i]$ *and tell me its value*
- L`.remove(`x`)` — *look for* x *in* L*; remove first copy found*
- L`.reverse()`
- L`.sort()` — *sort* L *according to "natural" order*
  a good idea only for "primitive" elements

Example

```
sage: my_list
[1, 5, 0, 5]

sage: my_list.extend((2,4))

sage: my_list
[1, 5, 0, 5, 2, 4]

sage: my_list.insert(3,-1)

sage: my_list
[1, 5, 0, -1, 5, 2, 4]

sage: my_list.pop(3)
-1

sage: my_list.sort()

sage: my_list
[0, 1, 2, 4, 5, 5]
```

A word on inserting

start:

my_list	1	5	0	5	2	4
	L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

```
sage:  my_list.insert(3,-1)
```

A word on inserting

start:

my_list	1	5	0	5	2	4
	L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

```
sage:  my_list.insert(3,-1)
```

A word on inserting

start:

my_list	1	5	0	5	2	4
	L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

```
sage:  my_list.insert(3,-1)
```

my_list	1	5	0	-1	5	2	4
	L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]

Nice dog! Does he do any tricks?

sets as Python tools

- S.`add`(x)
- S.`clear()` *remove all elements from* S
- S.`pop()` *removes and reports random (first?) element of* S
- S.`remove`(x) *remove* x *from* S

sets as mathematical objects

- S.`difference`(C) *difference* $S \setminus C$
- S.`intersection`(C) *intersection* $S \cap C$
- S.`union`(C) *union* $S \cup C$
- S.`isdisjoint`(C) `True` *iff* S *and* C *share no elements*
- S.`symmetric_difference`(x) *symmetric difference* $S \setminus C \cup C \setminus S$

Example

```
sage: my_set = set((1,5,0,5))

sage: my_set.add(4)

sage: my_set
set([0, 1, 4, 5])

sage: my_set.isdisjoint((-1,-2,4))
False

sage: my_set.symmetric_difference((-1,-2,4))
set([-2, -1, 0, 1, 5])

sage: my_set.remove(2)
...Output removed...
KeyError: 2

sage: my_set.remove(1)

sage: my_set
[0, 4, 5]
```

Outline

1 Collections in Python

2 Ranges of data

3 Strings

4 Summary

Tricks with []

Negative indices have meaning:

	L[0]	L[1]	L[2]	L[3]	L[4]
IndexError	1	5	0	5	IndexError
L[-5]	L[-4]	L[-3]	L[-2]	L[-1]	

Tricks with []

Negative indices have meaning:

	L[0]	L[1]	L[2]	L[3]	L[4]
IndexError	1	5	0	5	IndexError
L[-5]	L[-4]	L[-3]	L[-2]	L[-1]	

Example

```
sage:  L = [1,5,0,5]
sage:  L[-1]
5
sage:  L[-4]
1
sage:  L[-5]
```

. . . Output deleted. . .

```
IndexError:  list index out of range
```

Tricks with [:]

C[*first* : *last*+1] specifies subcollection

C[*first*]	C[*first*+1]	...	C[*last*]

- omit *first*? $\Longrightarrow$ start at `C[0]`
- omit *last*? $\Longrightarrow$ end at `C[-1]`

Tricks with [:]

C[*first* : *last*+1] specifies subcollection

C[*first*]	C[*first*+1]	...	C[*last*]

- omit *first*? $\Longrightarrow$ start at `C[0]`
- omit *last*? $\Longrightarrow$ end at `C[-1]`

Example

```
sage:  L[2:4]
[0, 5]
sage:  L[:2]
[1,5]
sage:  L[2:]
[0,5]
sage:  L[:]
[1,5,0,5]
```

`L[2]` *to* `L[3]`

`L[0]` *to* `L[1]`

`L[2]` *to* `L[-1]`

`L[0]` *to* `L[-1]`

The `range()` command

`range(` *first*, *last*+1 `)` generates a list with $n = last + 1 - first$ elements where

- *first* is the first integer in the list
  - default value is `0`
- *last* is the last integer in the list
- $first \geq last$? empty list

Example

```
sage: range(5)
[0, 1, 2, 3, 4]

sage: range(1,5)
[1, 2, 3, 4]

sage: range(3,5)
[3,4]

sage: range(5,5)
[]

sage: range(6,5)
[]
```

Outline

1 Collections in Python

2 Ranges of data

3 Strings

4 Summary

Strings

String: ordered collection of characters

'Hello' $\leftrightsquigarrow$

H	e	l	l	o

- extract elements using `[]`
- join elements using +
- other useful functions on pg. 96 of text

Example

```
sage: name = 'Euler'
sage: name[2]                    3rd character
'l'
sage: name[-1]                   last character
'r'
sage: name[0:4]                  first four characters in string
'Eule'
sage: name + ' computed'         add string; notice space
'Euler computed'
```

The `str()` command

`str(`x`)` where

- x is any object that can be turned into a string
- Sage will turn a *lot* of objects into strings!

Example

Numbers:

```
sage:  name + ' computed' + ' e**(i*pi) + 1 = '
         + str(0)
'Euler computed e**(i*pi) + 1 = 0'
```

Example

Numbers:

```
sage:  name + ' computed' + ' e**(i*pi) + 1 = '
         + str(0)
'Euler computed e**(i*pi) + 1 = 0'
```

Equations: (after "obvious" simplifications!)

```
sage:  name + ' computed ' + str(e**(i*pi) + 1 == 0)
'Euler computed 0 == 0'
```

Outline

1 Collections in Python

2 Ranges of data

3 Strings

4 Summary

Summary

- Through Python, Sage offers several kinds of collections
  - tuples, lists, sets
- Operations
  - `[]` for extraction
    - negatives allowed
    - `[:]` gives subcollections
  - usual mathematical operations on sets
  - others supplied by Python
- Strings allow lists of characters
  - `str(`x`)` produces "obvious" string representation of x