MAT 305: Mathematical Computing

Collections

John Perry

University of Southern Mississippi

Spring 2017

Outline

Collections?

Collection: group of objects identified as single object

- indexed
  - tuples $(a_0, a_1, a_2, \ldots a_n)$
    - points (x_0, y_0), (x_0, y_0, z_0)
  - lists $[a_0, a_1, \ldots, a_n]$
    - sequences $(a_0, a_1, a_2, \ldots)$
- not indexed
  - sets $\{a_0, a_5, a_3, a_2, a_1\}$
  - dictionaries

Outline

Python collections

Standard Python collections

- *indexable* or *ordered* (“sequence types”)
  - tuples, lists
  - access “element in position *i*” using `[i]`
    - but! start counting from `0`, **not 1**
- *not indexable* or *unordered* (“set types”)
  - sets, dictionaries
  - only one instance of any element
  - access *an* element, but not “element in position *i*”

Tuples

tuple: immutable, ordered collection

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

Example

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

```
sage: my_tuple[2]
0
```
access 3rd element (element 2)

```
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

list: 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 (element 2)

```
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 `{`*entries*`}`, `set(`*tuple or list*`)`, or `set()` (for empty set)
- redundant elements automatically deleted

Example

```
sage: my_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
set([0, 1, 5])
```
so what's in there, anyway?
not original list!

Dictionaries

A **dictionary** is a mutable, undordered collection

- *mutable*: can change elements
- *non-indexable*
  - cannot access elements by their order
  - computer arranges elements for efficiency
- defined using `dict(`*list of tuples*`)` or `{`d_1`:`a_1`,` d_2`:`a_2`, ...}`
  - entry d_i has the "meaning" a_i
- redundant elements automatically deleted

Example

```
sage: D = {1:3, 2:5}
```
dictionary w/2 entries

```
sage: D[1]
3
```
entry "1" has meaning 3

```
sage: D[0]
```
access element `0`*?*

…Output deleted…

```
KeyError: 0
```

Nice dog! Does any tricks? (1)

sets, dictionaries, tuples, and lists

- `type(`C`)`
  type of C
- `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 any tricks? (2)

lists — *these commands* ***change the list***

- L.append(x) — *add x at end of L*
- 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 any tricks? (3)

sets as Python tools — *these commands **change the set***

- 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 mathematics — *these commands **do not** change the set*

- S`.difference(`C`)` — *difference $S\backslash C$*
- 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(`C`)` — *symmetric difference $S\backslash C \cup C\backslash S$*

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]
```

Nice dog! Does any tricks? (4)

dictionaries — *these commands **change the dictionary***

- D`.clear()` — *remove all elements from D*
- D`.pop(`d`)` — *remove entry for d from D*
- D`.popitem()` — *remove random entry from D*
- D`.update(`C`)` — *add definitions in C to D*

*these commands **do not** change the dictionary*

- D`.keys()` — *list the keys (entries) of D*
- D`.values()` — *list the values (definitions) of D*

Outline

Arguments, lists and sets

- *Ordinarily*, function cannot change the value of an argument outside function
- However, if argument is a mutable collection C:
  - C cannot be changed, but
  - *elements* of C can be changed

Example: C does not change

```
sage: def modify_C(C):
        C = [0,1,2,3]
sage: L = [-1,0,1]
sage: modify_C(L)
sage: L
[-1, 0, 1]
```

Example: elements of C change

```
sage: def modify_els_of_C(C):
        C[0] = 0
sage: L = [-1,0,1]
sage: modify_els_of_C(L)
sage: L
[0, 0, 1]
```

Why does this happen?

Hand-waving / Lawyer's argument

- `L` is a list of 3 elements
    - data does not change
    - function concludes: `L` is still a list of 3 elements
- `L[0]`, `L[1]`, `L[2]` are *elements* of `L`
    - these data **are not** "arguments" to function
    - ∴ can be changed

Why does this happen?

Analogy: defacing library books doesn't change catalog

- `L` is address of a location in memory
    - similar to library's reference number for book

Why does this happen?

Analogy: defacing library books doesn't change catalog

- `L` is address of a location in memory
  - similar to library's reference number for book
- Python *copies* `L`'s value
  - write reference number on a scrap sheet of paper
  - original reference still in catalog

Why does this happen?

Analogy: defacing library books doesn't change catalog

- `L` is address of a location in memory
  - similar to library's reference number for book
- Python *copies* `L`'s value
  - write reference number on a scrap sheet of paper
  - original reference still in catalog
- Function learns (and cannot change) `L`'s value, *but*…
  - can deface book at that location, *even though*
  - changing number on scrap sheet of paper (`C`) doesn't change catalog entry (`L`)
  - ∴ function can change information at location

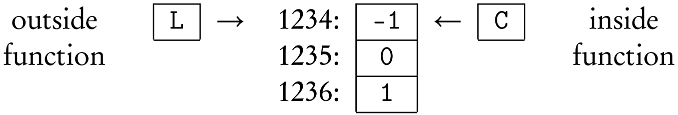
Why does this happen?

Analogy: defacing library books doesn't change catalog

- `L` is address of a location in memory
  - similar to library's reference number for book
- Python *copies* `L`'s value
  - write reference number on a scrap sheet of paper
  - original reference still in catalog
- Function learns (and cannot change) `L`'s value, *but*…
  - can deface book at that location, *even though*
  - changing number on scrap sheet of paper (`C`) doesn't change catalog entry (`L`)
  - ∴ function can change information at location
- Function concludes: data changed but `L` unchanged
  - books defaced, but catalog still references them

Why does this happen?

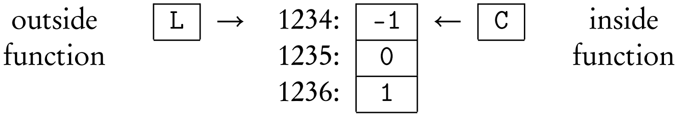
Precise answer: lists are pointers



- List @ location 1234 $\Longrightarrow$ `L` $\longrightarrow$ `1234`

Why does this happen?

Precise answer: lists are pointers



- List @ location 1234 $\Longrightarrow$ `L` $\longrightarrow$ `1234`
- $\therefore$ `C` $\longrightarrow$ `1234`
- Function now has access to memory *at* `L`
  - changing `C` won't change `L`
  - changing `C[0]` changes `L[0]`

Outline

A different sort of sort

Let's redefine our list:

```
sage: L = [1, 5, 0, 5, 3, 10, -3, 17, -10]
```

A different sort of sort

Let's redefine our list:

```
sage: L = [1, 5, 0, 5, 3, 10, -3, 17, -10]
```

Default sort:

```
sage: L.sort()
sage: L
[-10, -3, 0, 1, 3, 5, 5, 10, 17]
```

But what if I want to sort a different way?

Who cares?

Well ordering

An ordering of a set S is **well ordered** if every subset has a smallest element.

With the usual ordering $a < b$:

- $\mathbb{N}$ is well-ordered (**W**ell-**O**rdering **P**roperty)
- $\mathbb{Z}$ is not
  $\{0, -1, -2, -3, \ldots\}$ has no "minimum"

…but a different ordering might guarantee a minimum!

Restore `0` to its rightful place

Example

`0`, -1, 1, -2, 2, -3, 3, …

In this ordering of $\mathbb{Z}$:

- `0` "smallest"
- -1 next smallest
- 1 third smallest

…

Restore `0` to its rightful place

Example

`0`, -1, 1, -2, 2, -3, 3, …

In this ordering of $\mathbb{Z}$:

- `0` "smallest"
- -1 next smallest
- 1 third smallest

…

Order by **absolute** value first, *then* by value!

"Teach" Sage this ordering!

L`.sort(key=`*key_function*`)` where

- *key_function* maps L to an ordered set
- L's elements ordered according to this set

“Teach” Sage this ordering!

L.sort(key=*key_function*) where

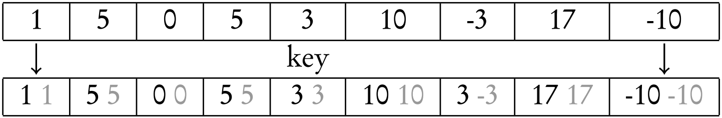
- *key_function* maps *L* to an ordered set
- *L*’s elements ordered according to this set

```
sage:  def by_absolute_value(n):
           return abs(n)
sage:  L = [1, 5, 0, 5, 3, 10, -3, 17, -10]
sage:  L.sort(key=by_absolute_value)
sage:  L
[0, 1, 3, -3, 5, 5, 10, -10, 17]
```

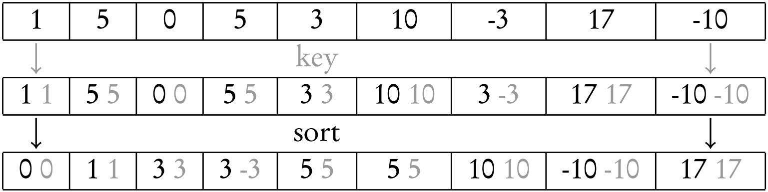
What happened?

1	5	0	5	3	10	-3	17	-10

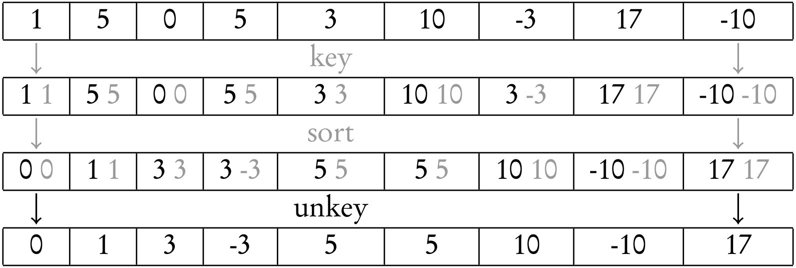
What happened?



What happened?



What happened?



And if we want to refine the ordering further?

```
sage: L
[0, 1, 3, -3, 5, 5, 10, -10, 17]
```

What if we want $\ldots, -3, 3, \ldots, -10, 10, \ldots$ instead?

And if we want to refine the ordering further?

```
sage: L
[0, 1, 3, -3, 5, 5, 10, -10, 17]
```

What if we want $\ldots, -3, 3, \ldots, -10, 10, \ldots$ instead?

Refine with tuples!

```
sage: def by_absolute_value_negatives_first(n):
          return (abs(n), n)
sage: L = [1, 5, 0, 5, 3, 10, -3, 17, -10]
sage: L.sort(key=by_absolute_value_negatives_first)
sage: L
[0, 1, -3, 3, 5, 5, -10, 10, 17]
```

Outline

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*? ⟹ start at C[0]
- omit *last*? ⟹ 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 list w/*last* + 1 - *first* elements

- *first* indexes the first element
  - default value is `0`
- *last* indexes the last element
- *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

Strings

String: ordered collection of characters

'Hello' ⇝

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]
'l'
sage: name[-1]
'r'
sage: name[0:4]
'Eule'
sage: name + ' computed'
'Euler computed'
```

3rd character

last character

first four characters in string

add string; notice space

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

Summary

- Through Python, Sage offers several kinds of collections
  - tuples, lists, sets, dictionaries
- 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