

# Midterm Review 2023

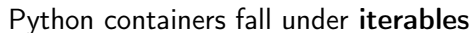
Jean Mark Gawron

bda/ling 572  
San Diego State University

February 28, 2023

## Section 1

# Python Types: Containers



## Section 2

# Containers

## String example

Strings are containers. What do they contain?

```
>>> X = 'abcde'
>>> 'c' in X      # X contains element 'c'
True
>>> len(X)        # How many elements does X have?
5
```

Every container supports the **in** test. Every container has a computable number of elements (**len** function).

# Most important Python type concepts

- Iterables: Can be looped through. All containers are iterables; on the Midterm we focus on containers.
- Containers: Iterables which have length and support the `in` test)
  - ① Sequences (as Containers have length, support `in`)
    - ① Tuples, Lists, Strings (there are other builtin Python sequence types, but not on this midterm).
    - ② Sequences contain data in a fixed order.
    - ③ Data can be indexed by position (integers).
    - ④ Indexing data by a non-integer is a `TypeError`.
  - ② Non-Sequence Containers (Sets and Dictionaries)
    - ① Dictionary is a Mapping;  $\text{key} \mapsto \text{value}$
    - ② Set: Unordered, no duplicates

## List example

Lists can contain just about any data type

```
>>> L = [24, 3.14, 'w', [1, 'a']]
>>> len(L)                                # 4 items
4
>>> 24 in L                               # container supports in test
True
>>> L[2]                                  # Indexing by position
'w'
>>> [1, 'a'] in L                         # This list contains a list
True
>>> Z = []                               # empty list, length 0
```

# General features

- 1 Every container has an empty version
- 2 Every container has a delimiter that can be used when listing elements
- 3 Every type can be used as a function to create new instances

Type	Delimiter	Empty	Function	
list	[...]	[]	list('abc')	['a', 'b', 'c']
tuple	(...)	()	tuple('abc')	('a', 'b', 'c')
string	"..."	""	str('abc')	'abc'
dictionary	{key : val, ... }	{ }	dict(...)	later
set	{... }	set()	set('abc')	{ 'c', 'b', 'a' }



## Sequences as ordered data

```
>>> L
[24, 3.14, 'w', [1, 'a']]
>>> L[0]      # 1st element, 0-based indices
24
>>> L[1]      # 2nd element
3.14
>>> L[-1]     # last element
[1, 'a']
>>> L[4]      # Raises IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Note: Everything on this slides works for all sequences, not just lists

# Sequence slices

```
>>> L
[24, 3.14, 'w', [1, 'a']]
>>> L[1:3] # sublist of 2nd and 3rd elements
[3.14, 'w']
>>> L[:-1] # sublist excluding last element
[24, 3.14, 'w']
```

$S[m:n]$ : a subsequence containing  $n-m$  elements, starting at  $S[m]$  and going up through  $S[n-1]$ :

## Sequence slices II

All sequence indexing works the same

```
>>> T = (23, -1, 'September')
>>> G = 'lawyer'
>>> T[0:2] # Tuple of 1st and 2nd elements
(23, -1)
>>> G[0:2] # T[0:2] # String with 1st and 2nd chars
'la'
>>> T[: -1] # excluding last element
(23, -1)
>>> G[: -1] # excluding last char
'lawye'
```

# Dictionaries are mappings I

Represent a systematic mapping **from** one kind of data (keys) **to** another (values).

New York	8,175,133	{'New York':	8175133,
Los Angeles	3,792,621	'Los Angeles':	3792621,
Chicago	2,695,598	'Chicago':	2695598,
Houston	2,099,451	'Houston':	2099451
		}	

## Dictionaries II

```
>>> dd1 = dict([('a',23),('b',-1),('f','September')])
>>> dd1
{'a': 23, 'b': -1, 'f': 'September'}
>>> dd = {'a': 23, 'b': -1, 'f': 'September'}
>>> dd['f']      # element associated with key 'f'
'September'
>>> dd['c']      # Raises KeyError (no IndexError)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

Data is accessed by key, not by a numerical index. Not sequences. In classic python, there is no notion of order in a dictionary.

## Sets: No order, no duplicates

```
>>> ss = {23, 'b', 'September'}
>>> ss.add('c')
>>> ss          # can add elements
{'b', 'c', 'September', 23}
>>> ss.add('c')
>>> ss          # no effect, no duplicates
{'b', 'c', 'September', 23}
>>> ss.remove('c') # Can remove by key
>>> ss
{'b', 'September', 23}
>>> ss.remove('d')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'd'

# Assigning Values/Updating

```
>>> result = ['January', 'February', 'March', 'April']
>>> result[1] = 'September' # 2nd element -> 'September'
>>> result
['January', 'September', 'March', 'April']
>>> result[0:2] = ['December']
>>> result # First 2 elements changed
['December', 'March', 'April']
```

Expression on the left of = is the same expression that is used for indexing the container. Expression on the right is the new value creating/overwriting data at that index.

## Assigning Values Dictionary

```
>>> dd
{'a': 23, 'b': -1, 'f': 'September'}
>>> dd['a'] = 25
>>> dd                                     # New value for old key
{'a': 25, 'b': -1, 'f': 'September'}
>>> dd['e'] = 22
>>> dd                                     # New key, new value
{'a': 25, 'b': -1, 'f': 'September', 'e': 22}
```

Expression on the left of = is the same expression that is used for indexing a dictionary. Expression on the right is the new value creating/overwriting data at that index.



# Immutability

```
>>> G = 'back'
>>> G[0] = 's'      # Strings are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Trying to assign to/update an immutable type is a `TypeError`

# Mutability Summary

Mutable	Immutable
list	tuple
dict	??
set	frozenset
??	str

Trying to assign to/update an immutable type is a `TypeError`

# Summary of basic container properties

Type	Sequence?	Mutable?	Contents
list	yes	yes	unrestricted
tuple	yes	no	unrestricted
string	yes	no	$i \text{ in string} \Rightarrow i \text{ is a string}$
set	no	yes	immutable
frozenset	no	<b>no</b>	immutable
dict	no	yes	key must be immutable

## Section 3

# Loops, Functions, Conditionals

# Loops

```
>>> L2 = ['zaza', 'baba', 'dcdc', 'mlml']
>>> res1 = [ ]
>>> for x in L2:
...     if x not in ['baba', 'cbcb']:
...         res1.append(x)
...
>>> res1
['zaza', 'dcdc', 'mlml']
>>> res2 = [x for x in L2 if x not in ['baba', 'cbcb']]
>>> res2
['zaza', 'dcdc', 'mlml']
```

# Functions

```
>>> [x for x in L2 if x not in ['baba', 'cbcb']]
['zaza', 'dcdc', 'mlml']
>>> def filter_from(Seq1, Seq2):
...     return [x for x in Seq1 if x not in Seq2]
...
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> wings = ['paul', 'linda', 'denny']
>>> res3 = filter_from(beatles, wings)
>>> res3
['john', 'george', 'ringo']
```

# Function calls

Define a function `filter_from_upper` that upper cases all the elements of a sequence that do not belong to a second given sequence. It should call `filter_from`.

```
>>> def filter_from_upper(Seq1, Seq2):  
...     return [x.upper() for x in filter_from(Seq1, Seq2)]  
...  
>>> beatles = ['john', 'paul', 'george', 'ringo']  
>>> wings = ['paul', 'linda', 'denny']  
>>> res4 = filter_from_upper(beatles, wings)  
>>> res4  
['JOHN', 'GEORGE', 'RINGO']
```

# Loops and function concepts

- Loops
  - ① for-loops
  - ② list-comprehensions
- Functions
  - ① Function Definition Syntax

```
def FunctionName (Param1, Param2, ...):
```

```
    Line1
```

```
    Line2
```

```
    return PythonExpression
```

```
    ...
```

```
    Linen
```

- ② Calling a function from inside another function



# Conditionals

X is an integer

```
>>> X = 5
>>> if X > 6:
...     result = 'Greater than 6'
... elif X > 5:
...     result = 'X = 6!'
... else:
...     result = 'Less than 6'
...
>>> result
'Less than 6'
```

## Section 4

# Numpy

# 1- and 2-D arrays

```
>>> import numpy as np
>>> a1d = np.arange(12)
>>> a1d
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a2d = np.arange(12).reshape((3,4))
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

# Indexing 1- and 2-D arrays

```
>>> a1d
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a1d[3]      # 1D array, 1 index suffices
3
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a2d[2,1]    # 2D array: [row,col] index
9
```

## Slicing 1- and 2-D arrays

```
>>> a1d
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a1d[2:4]
array([2, 3])
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a2d[1:3,:]
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a2d[:,1:3]
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

# Broadcasting

```
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> 2* a2d    # On midterm
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]])

>>> a2d * np.array([1,2,3,4])    # Weight each col differently
array([[ 0,  2,  6, 12],
       [ 4, 10, 18, 28],
       [ 8, 18, 30, 44]])
```

Tricky broadcasting not on midterm

# Elementwise arithmetic

```
>>> x = np.arange(8).reshape((2,4))
>>> x
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> y = np.arange(0,16,2).reshape((2,4))
>>> y
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14]])
>>> x + y    # On midterm
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21]])
```

# Boolean arrays

```
>>> a1d
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a1d > 5           # Broadcasting happens
array([False, False, False, False, False, False,  True,  True,
        True,  True,  True])
>>> a1d[a1d > 5]      # Using the Boolean array to index a1d
array([ 6,  7,  8,  9, 10, 11])
>>> a2d[a2d > 7]
array([ 8,  9, 10, 11])
```

Using the Boolean array for indexing is called **masking**



## Masking 2D arrays to select rows

```
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> a2d[:,2] > 5    # Applying a Boolean test to col2 of a2d
array([False,  True,  True])

>>> a2d[a2d[:,2] > 5,:] # Rows with col2 val > 5
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Placing a Boolean constraint on a col of a 2D array creates a Boolean array of the right length to select rows

# Boolean connectives I

```
>>> a2d
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> a2d[a2d[:,1] < 9]    # Rows with col1 val < 9
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

>>> a2d[a2d[:,2] > 5,:]  # Rows with col2 val > 5
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> a2d[(a2d[:,1] < 9) & (a2d[:,2] > 5)] # Combining masks
array([[4, 5, 6, 7]])
```

## Boolean connectives II

Given

```
bc1 = a2d[:,2] > 5
```

```
bc2 = a2d[:,1] < 9
```

We can create the following Boolean combinations:

$bc1 \ \& \ bc2$

$bc1$  and  $bc2$

$bc1 \ | \ bc2$

$bc1$  or  $bc2$

$\sim bc1$

not  $bc1$

$\sim bc1 \ \& \ bc2$

$bc2$  and not  $bc1$

$\sim (bc1 \ | \ bc2)$

neither  $bc1$  nor  $bc2$

# Data with numpy arrays

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> features = data['data']
>>> target = data['target']
```

Note that it's not necessary to import numpy to use the arrays defined in the dataset.

## Iris data: 1- and 2D array

```
>>> target[:5]      # Gives species 0 or 1 or 2 of each iris
array([0, 0, 0, 0, 0])
>>> features[:5,:]  # Gives 4 measurements for each iris
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

## Data features

```
>>> data.feature_names[:2]      # Names of first two cols
['sepal length (cm)', 'sepal width (cm)']
>>> data.feature_names[2:]      # Names of last two cols
['petal length (cm)', 'petal width (cm)']
>>> data.target_names          # Names for species vals 0,1,2
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
>>> (features[:,2] > 1.5)[:5]
array([False, False, False, False, False])
>>> features[(features[:,2] > 1.5)][:5]
array([[5.4, 3.9, 1.7, 0.4],
       [4.8, 3.4, 1.6, 0.2],
       [5.7, 3.8, 1.7, 0.3],
       [5.4, 3.4, 1.7, 0.2],
       [5.1, 3.3, 1.7, 0.5]])
```

## Section 5

# Odds and Ends

# Functions/Methods

- ❶ for x in range(5):  
    x = 0, 1, 2, 3, 4
- ❷ String methods: .title(), .lower(), .upper(), .split(), .join()
- ❸ Dictionary methods: .keys(), .values(), .items()
- ❹ Operators:
  - + (numbers, sequences), in (containers on the RHS)
  - Boolean operators: ==, <, >,



# Gotchas: String Operators, String looping

```
>>> 'aa' < 'ab'
```

```
True
```

```
>>> 'ab' < 'aa'
```

```
False
```

```
>>> 'ab' < 'ab'
```

```
False
```

```
>>> 'aa' + 'b' + 'dd'
```

```
'aabdd'
```

```
>>> ['a' + x.title() for x in 'bcde']
```

```
['aB', 'aC', 'aD', 'aE']
```

```
>>> ['a' + x.title() for x in ['bc', 'de', 'fg']]
```

```
['aBc', 'aDe', 'aFg']
```