# Structured Data

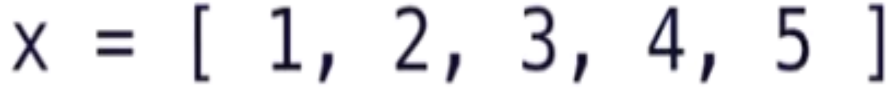
.

# Overview of Data Structures

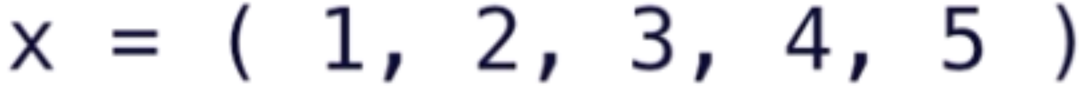
| | |
|---|---|
| • Lists | [ ] |
| • Tuples | ( ) |
| • Dictionaries | { *key : value* } |
| • Sets | { } |
| • List Comprehensions | *range*() |
| • Mixed Structures_ | |

# Basic Data Structures – Lists, Tuple

- List is a type of basic sequence using squared brackets
- Is mutable
- Tuple is like list, but immutable
- Uses parentheses



```
x = [ 1, 2, 3, 4, 5 ]
```



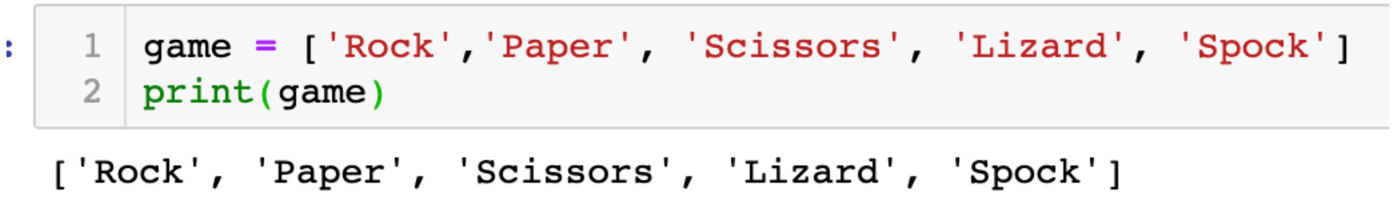
```
x = ( 1, 2, 3, 4, 5 )
```

# Python Lists

- Very versatile
- Denoted by brackets, elements separated by commas
- Index for the first element starts from 0 (Zero)
- Actions can be performed like *append*, *concatenate*, *sort*, *insert* and also *delete* elements from the list

# Python Lists

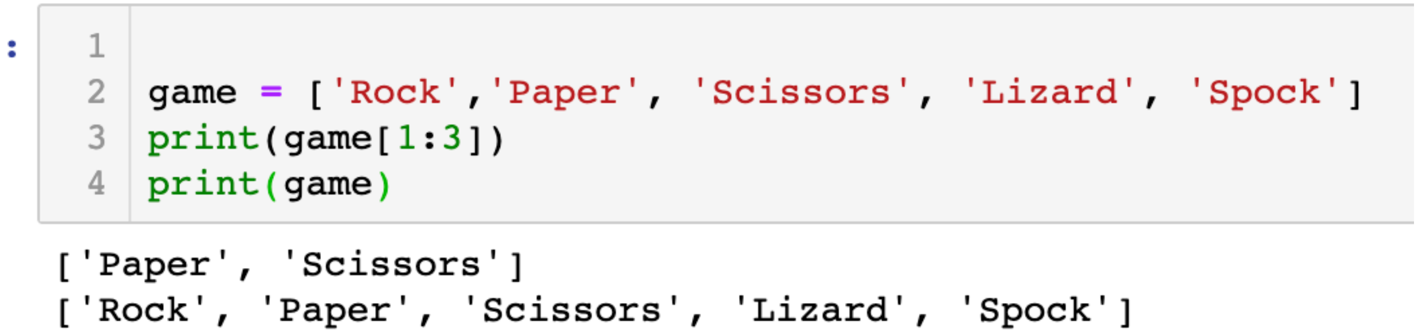
- Here we define a list with squared brackets



```
game = ['Rock','Paper', 'Scissors', 'Lizard', 'Spock']
print(game)
```

```
['Rock', 'Paper', 'Scissors', 'Lizard', 'Spock']
```

- Can print the whole list of the just the required ones by stating their index values



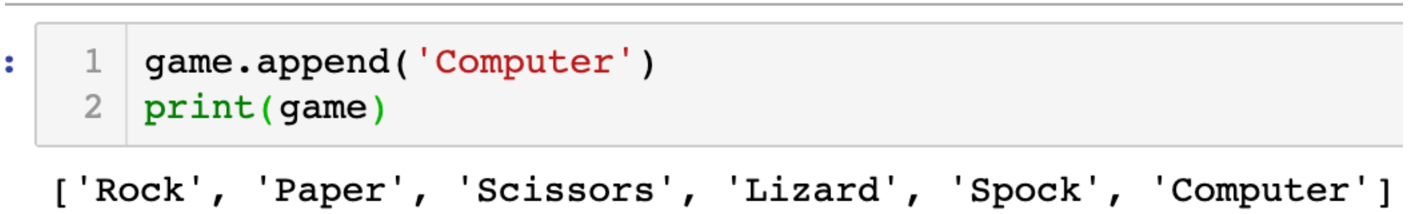
```

game = ['Rock','Paper', 'Scissors', 'Lizard', 'Spock']
print(game[1:3])
print(game)
```

```
['Paper', 'Scissors']
['Rock', 'Paper', 'Scissors', 'Lizard', 'Spock']
```

# Python Lists

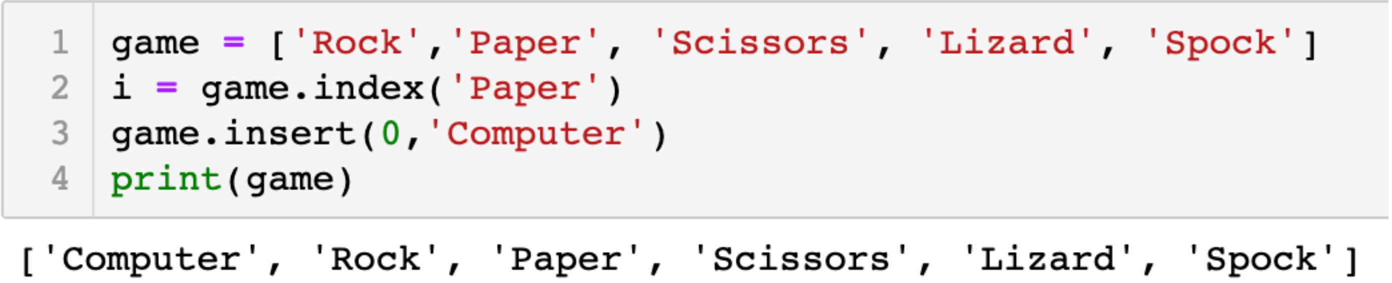
- Can append items in the list



```python
game.append('Computer')
print(game)
```

```
['Rock', 'Paper', 'Scissors', 'Lizard', 'Spock', 'Computer']
```

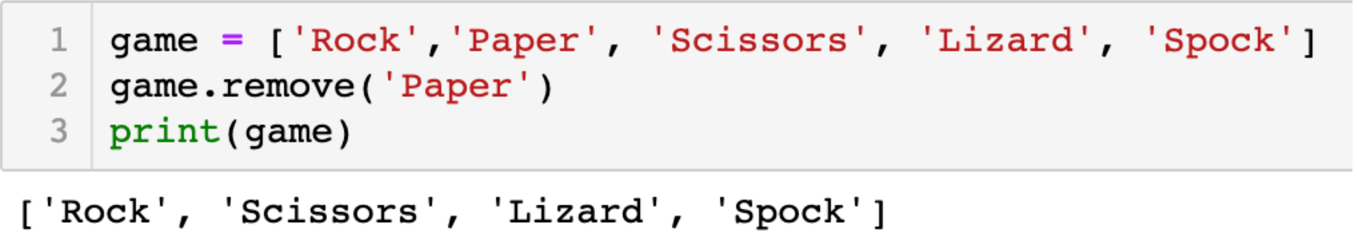
- Can insert items in the list



```python
game = ['Rock','Paper', 'Scissors', 'Lizard', 'Spock']
i = game.index('Paper')
game.insert(0,'Computer')
print(game)
```

```
['Computer', 'Rock', 'Paper', 'Scissors', 'Lizard', 'Spock']
```

- Can remove or delete items from the list

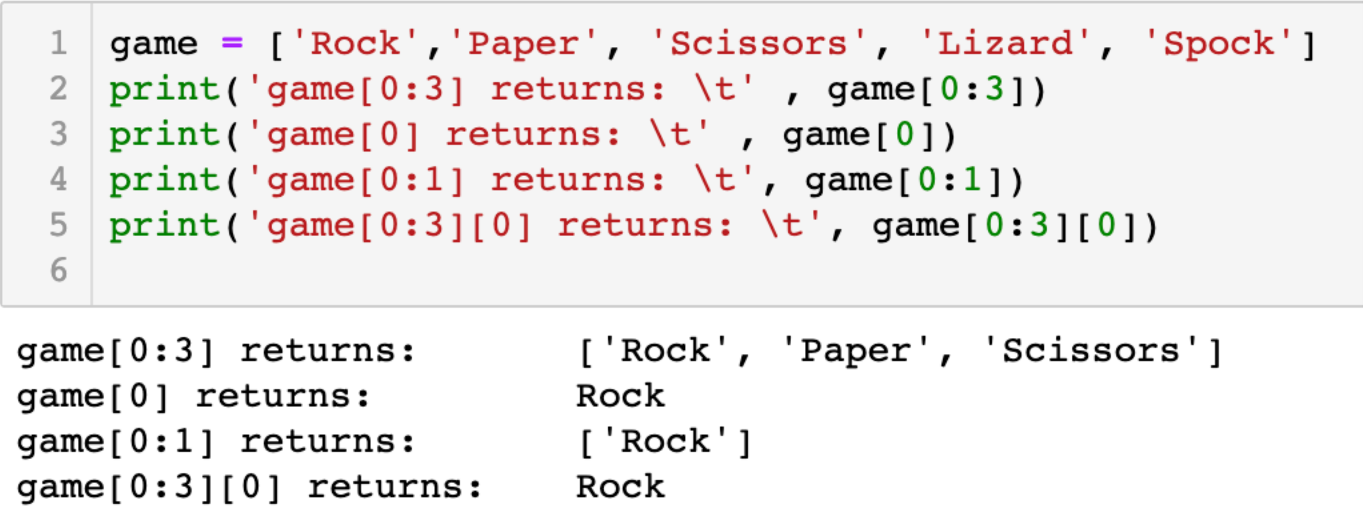


```python
game = ['Rock','Paper', 'Scissors', 'Lizard', 'Spock']
game.remove('Paper')
print(game)
```

```
['Rock', 'Scissors', 'Lizard', 'Spock']
```

# Slicing a List

- Collective elements in contiguous groups are known as slices
- E.g., If we wish to have a sub-list from a list, it can be written as a slice.
- The number of elements selected in a slice, is given by the difference of the two indices.
- Slicing always returns another list even if the list size is 1



```
game = ['Rock','Paper', 'Scissors', 'Lizard', 'Spock']
print('game[0:3] returns: \t' , game[0:3])
print('game[0] returns: \t' , game[0])
print('game[0:1] returns: \t', game[0:1])
print('game[0:3][0] returns: \t', game[0:3][0])

```

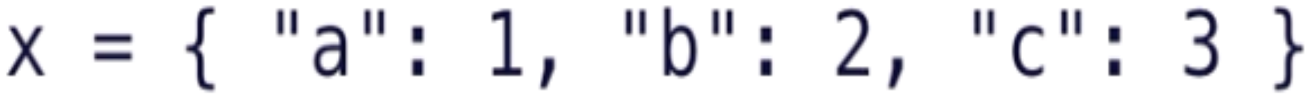
```
game[0:3] returns:          ['Rock', 'Paper', 'Scissors']
game[0] returns:            Rock
game[0:1] returns:          ['Rock']
game[0:3][0] returns:       Rock
```

# Python Lists are useful and ubiquitous

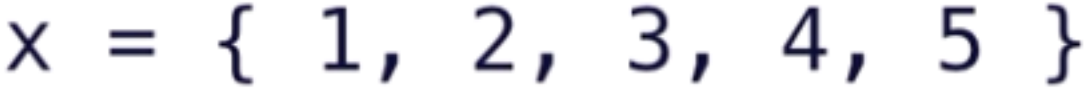
- Flexible and efficient containers
- Use where order of element matters
- Use the slicing syntax to operate on sub-lists
- Can be used for heterogeneous data (may contain different types of data) BUT mostly used for collecting data items of the same type

# Basic Data Structures – Dictionaries, Sets

- Dictionary is a sequence of key-value pairs
- Uses curly brackets
- Sets are unordered list of unique values
- Uses curly brackets



```
x = { "a": 1, "b": 2, "c": 3 }
```

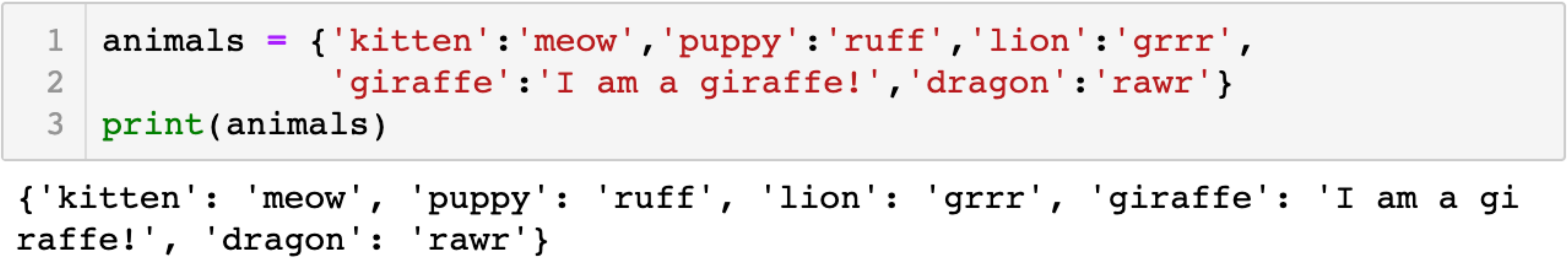


```
x = { 1, 2, 3, 4, 5 }
```

# Python Dictionaries

- Lists: Associate numerical indices, starting at zero with an order sequence of elements
- Dictionaries: Map names to values
- Any Python object that is hashable (i.e. can be converted to a number), can be used as a name.

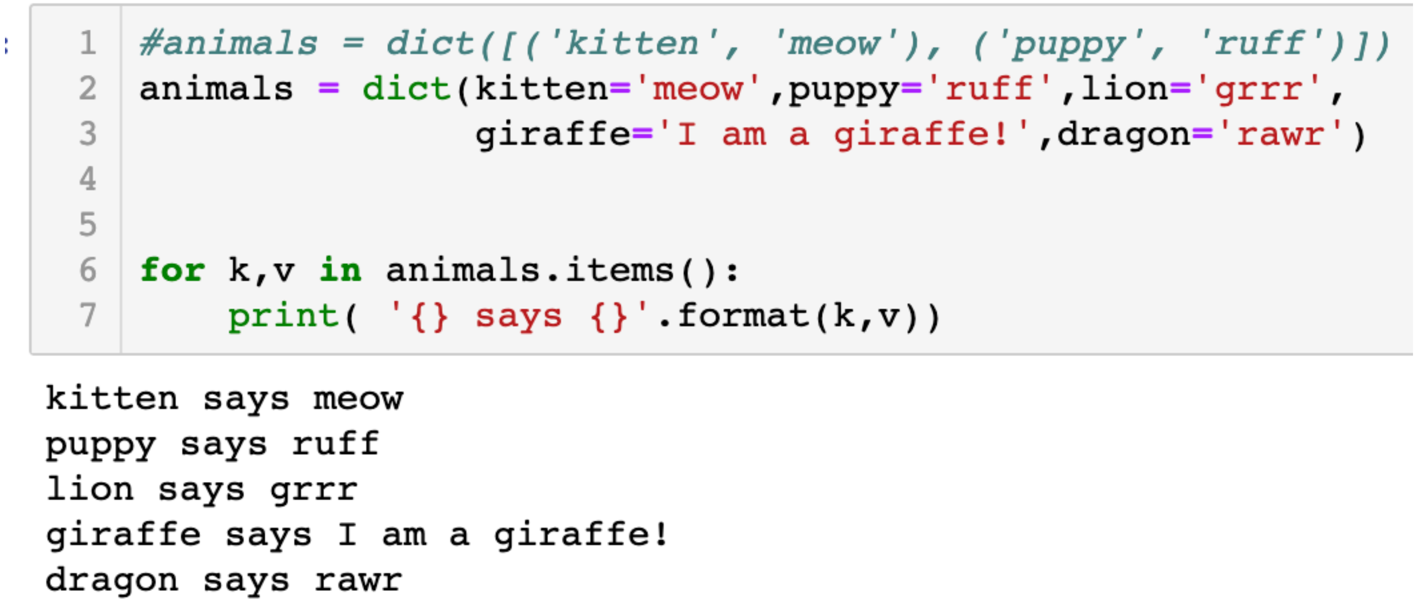
# Python Dictionaries



```
animals = {'kitten':'meow','puppy':'ruff','lion':'grrr',
           'giraffe':'I am a giraffe!','dragon':'rawr'}
print(animals)
```

```
{'kitten': 'meow', 'puppy': 'ruff', 'lion': 'grrr', 'giraffe': 'I am a gi
raffe!', 'dragon': 'rawr'}
```

- Dictionary types have hashed key value pairs
- Key and its value is separated by colon



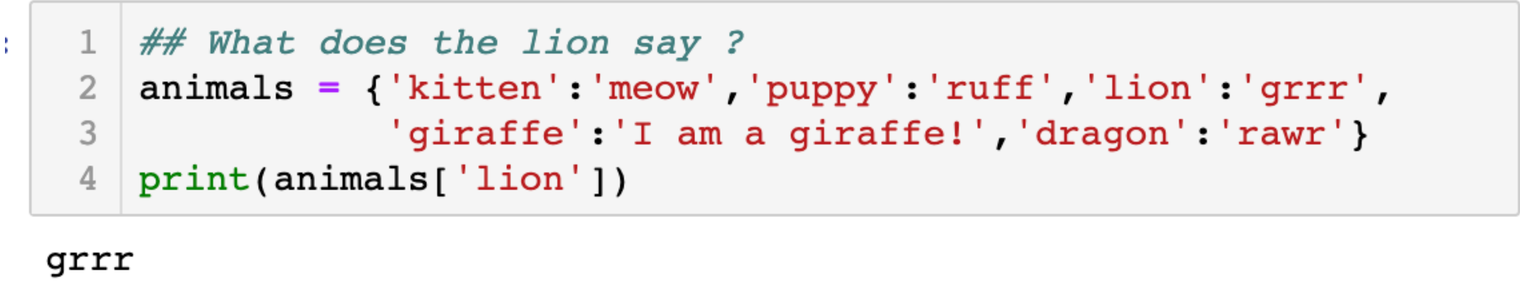
```
#animals = dict([('kitten', 'meow'), ('puppy', 'ruff')])
animals = dict(kitten='meow',puppy='ruff',lion='grrr',
               giraffe='I am a giraffe!',dragon='rawr')


for k,v in animals.items():
    print( '{} says {}'.format(k,v))
```

```
kitten says meow
puppy says ruff
lion says grrr
giraffe says I am a giraffe!
dragon says rawr
```

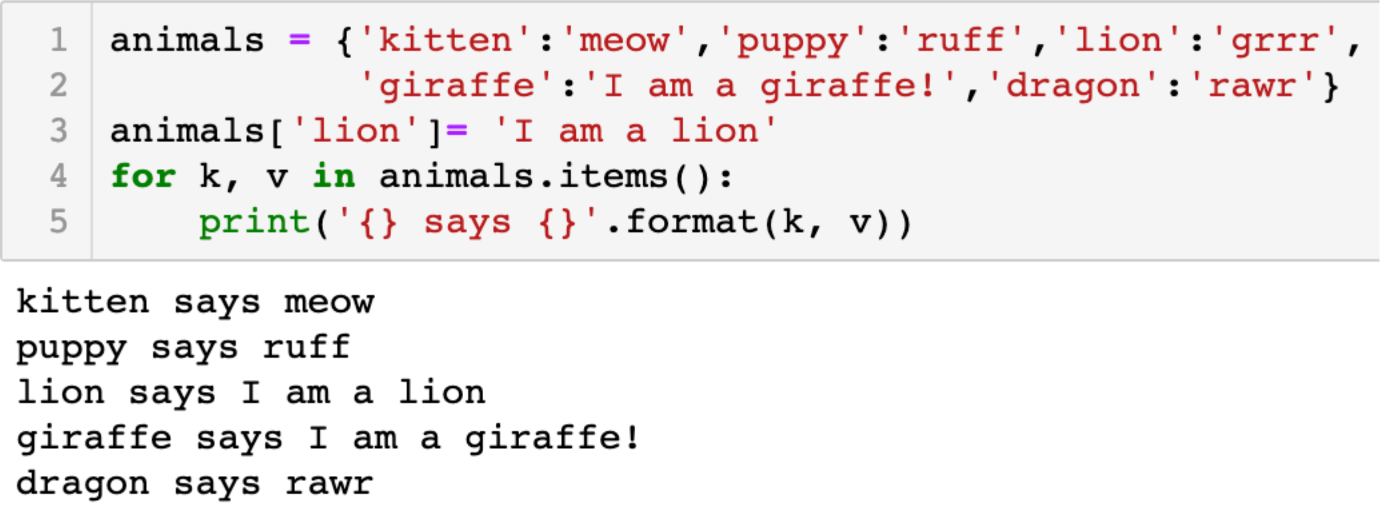
# Python Dictionaries

- Dictionary is indexed by its key.
- Easy to pick a particular element
- Can also add new items or update an existing value



```python
## What does the lion say ?
animals = {'kitten':'meow','puppy':'ruff','lion':'grrr',
           'giraffe':'I am a giraffe!','dragon':'rawr'}
print(animals['lion'])
```

```
grrr
```

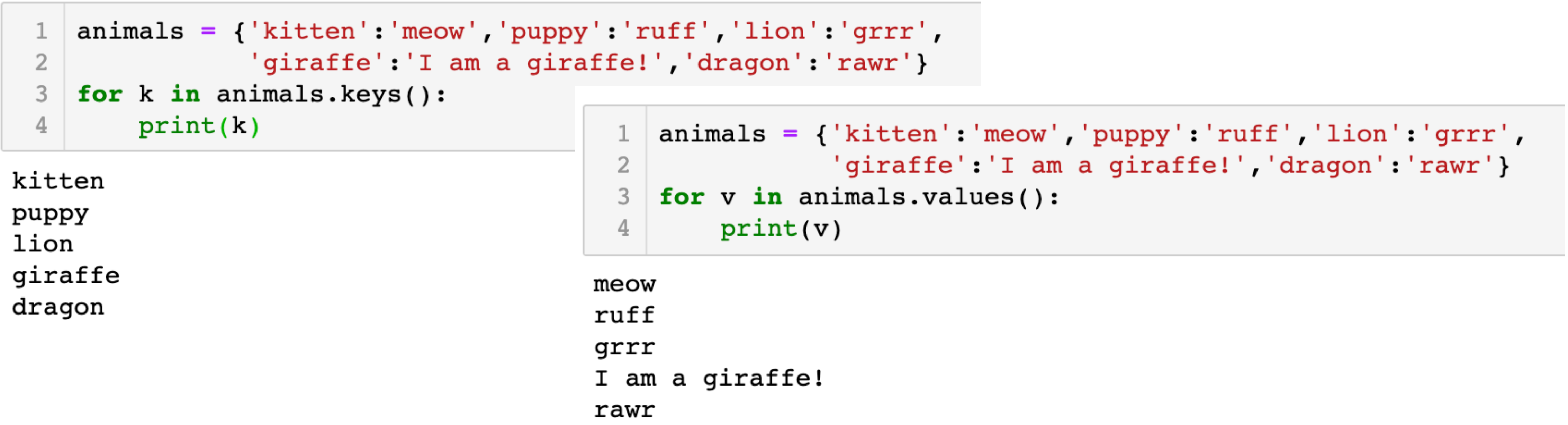


```python
animals = {'kitten':'meow','puppy':'ruff','lion':'grrr',
           'giraffe':'I am a giraffe!','dragon':'rawr'}
animals['lion']= 'I am a lion'
for k, v in animals.items():
    print('{} says {}'.format(k, v))
```

```
kitten says meow
puppy says ruff
lion says I am a lion
giraffe says I am a giraffe!
dragon says rawr
```

# Python Dictionaries keys and values

- Both dictionaries and values can be iterated



```python
animals = {'kitten':'meow','puppy':'ruff','lion':'grrr',
           'giraffe':'I am a giraffe!','dragon':'rawr'}
for k in animals.keys():
    print(k)
```

```
kitten
puppy
lion
giraffe
dragon
```

```python
animals = {'kitten':'meow','puppy':'ruff','lion':'grrr',
           'giraffe':'I am a giraffe!','dragon':'rawr'}
for v in animals.values():
    print(v)
```

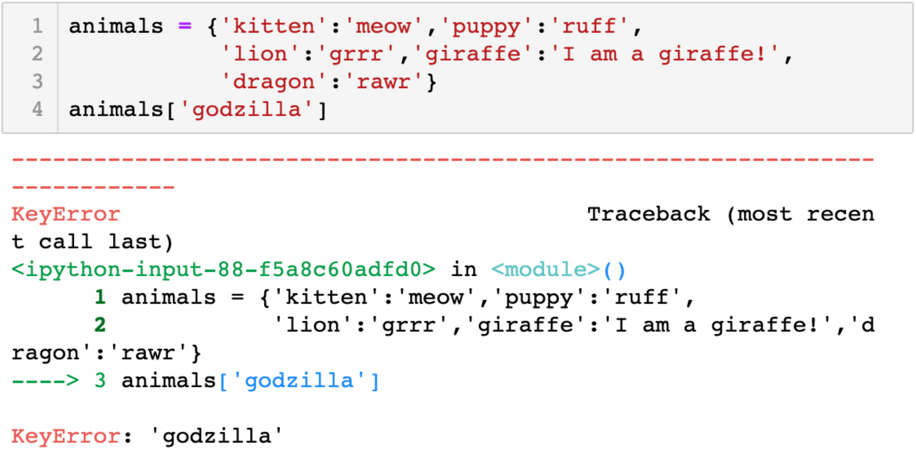
```
meow
ruff
grrr
I am a giraffe!
rawr
```

# Python Dictionaries key based calls

- Keys based calls require that key is in the dictionary otherwise we get ‘KeyError’

- You can first check if key in the dictionary

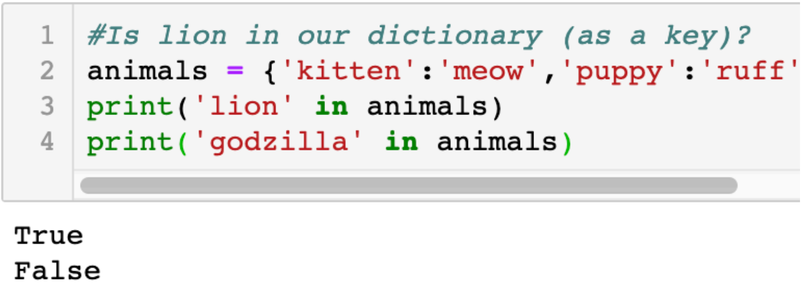
- Or use get function and provide an output for missing key



```
animals = {'kitten':'meow','puppy':'ruff',
           'lion':'grrr','giraffe':'I am a giraffe!',
           'dragon':'rawr'}
animals['godzilla']
```

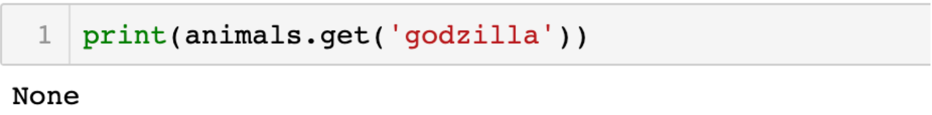
```
---------------------------------------------------------------------------
-------------
KeyError                                  Traceback (most recen
t call last)
<ipython-input-88-f5a8c60adfd0> in <module>()
      1 animals = {'kitten':'meow','puppy':'ruff',
      2            'lion':'grrr','giraffe':'I am a giraffe!','d
ragon':'rawr'}
----> 3 animals['godzilla']

KeyError: 'godzilla'
```



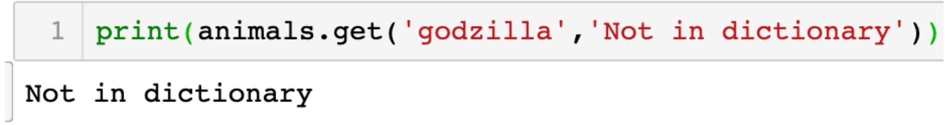
```
#Is lion in our dictionary (as a key)?
animals = {'kitten':'meow','puppy':'ruff'
print('lion' in animals)
print('godzilla' in animals)
```

```
True
False
```



```
print(animals.get('godzilla'))
```

```
None
```



```
print(animals.get('godzilla','Not in dictionary'))
```

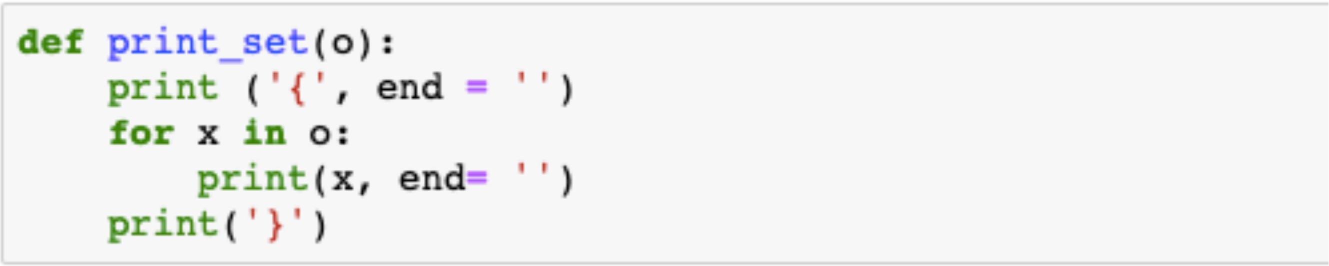
```
Not in dictionary
```

# Python Dictionaries are useful and ubiquitous

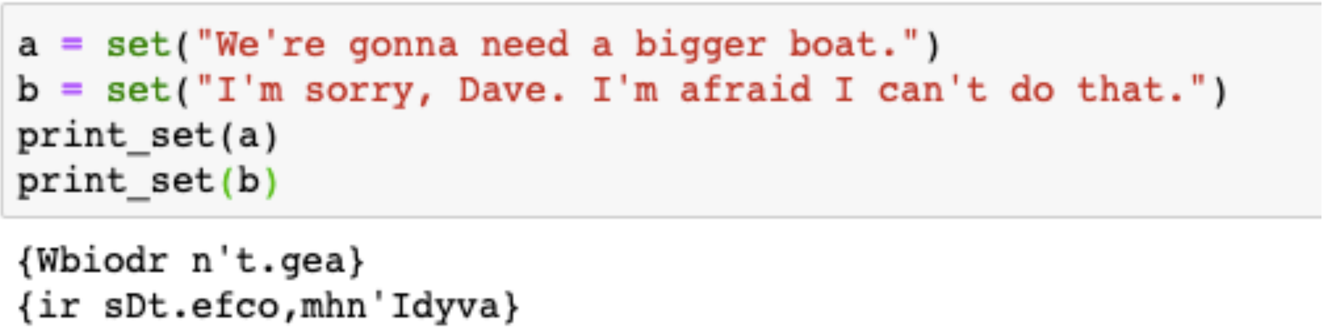
- Flexible and efficient containers for heterogeneous data
- Use where data items have/ can be given labels
- Most appropriate for collecting data items of different kinds

# Python Sets

- Set is like a list with unique elements
- You get an unordered list of unique characters in each string
- The result is different every time as it is unordered list

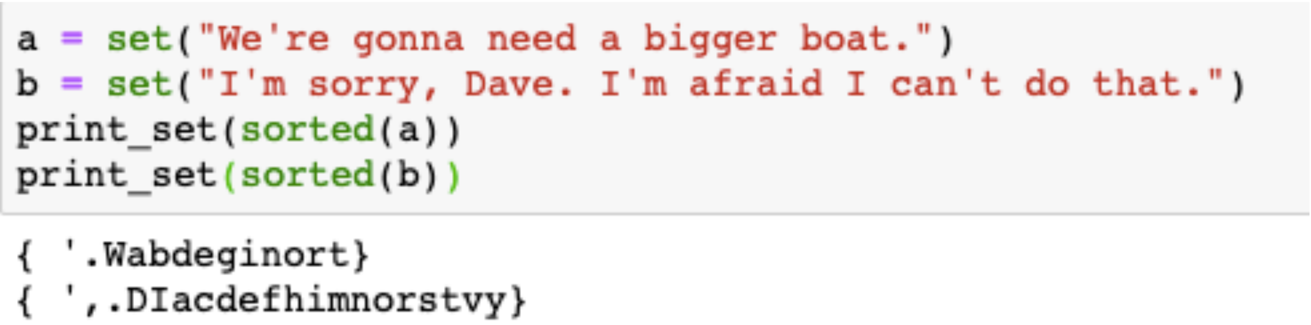


```
def print_set(o):
    print ('{', end = '')
    for x in o:
        print(x, end= '')
    print('}')
```



```
a = set("We're gonna need a bigger boat.")
b = set("I'm sorry, Dave. I'm afraid I can't do that.")
print_set(a)
print_set(b)
```

```
{Wbiodr n't.gea}
{ir sDt.efco,mhn'Idyva}
```

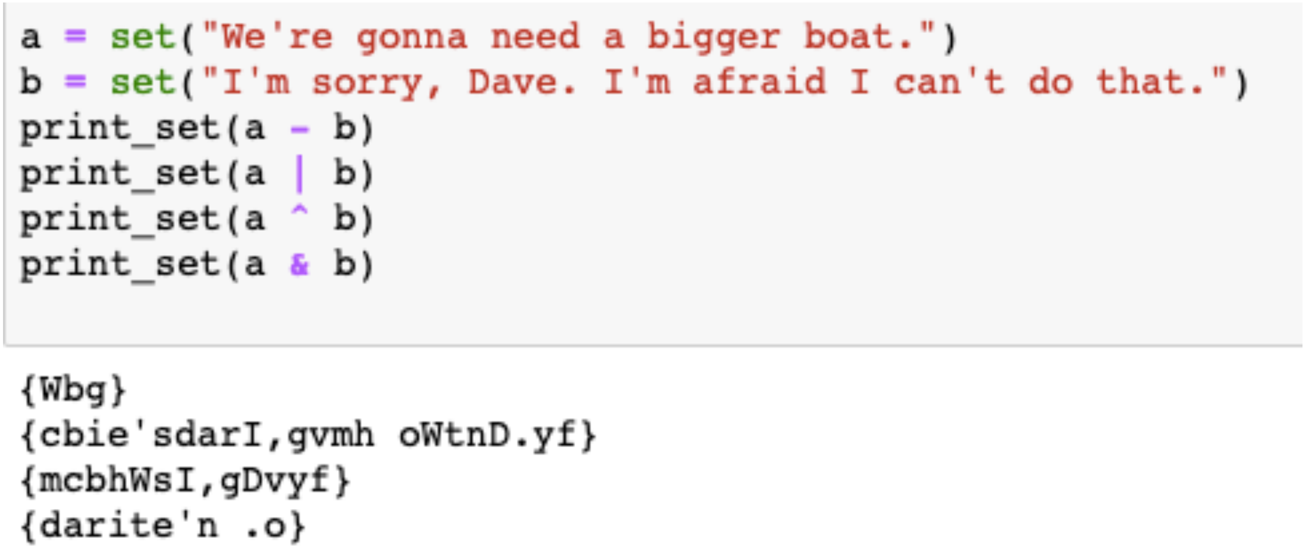


```
a = set("We're gonna need a bigger boat.")
b = set("I'm sorry, Dave. I'm afraid I can't do that.")
print_set(sorted(a))
print_set(sorted(b))
```

```
{ '.Wabdeginort}
{ ',.DIacdefhimnorstvy}
```

# Python Sets

- Can also use operators to check the members that are in a set



```
a = set("We're gonna need a bigger boat.")
b = set("I'm sorry, Dave. I'm afraid I can't do that.")
print_set(a - b)
print_set(a | b)
print_set(a ^ b)
print_set(a & b)
```

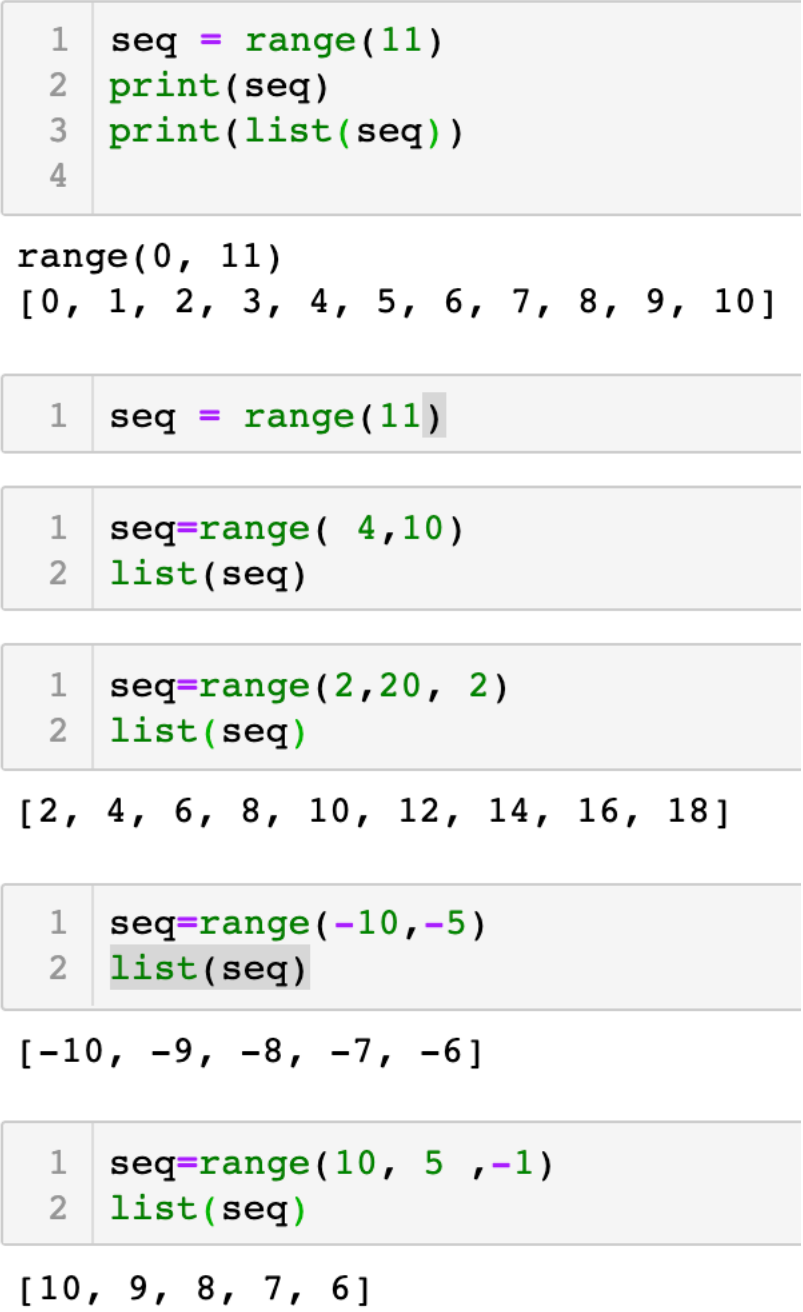
```
{Wbg}
{cbie'sdarI,gvmh oWtnD.yf}
{mcbhWsI,gDvyf}
{darite'n .o}
```

# List Comprehensions

- Comprehension feature can be used for iterations, performing operations on each element and collect results in a new list or dict.
- Requires shorter and cleaner code than loops.

# List comprehension

- List comprehension is a list created based on another list or iterator
- Created a sequence based on range function, from 0 to 10
- Created sequence 2 which includes items in seq1 multiplied by 2



```
seq = range(11)
print(seq)
print(list(seq))

```

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

```
seq = range(11)
```

```
seq=range( 4,10)
list(seq)
```

```
seq=range(2,20, 2)
list(seq)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
seq=range(-10,-5)
list(seq)
```

```
[-10, -9, -8, -7, -6]
```

```
seq=range(10, 5 ,-1)
list(seq)
```

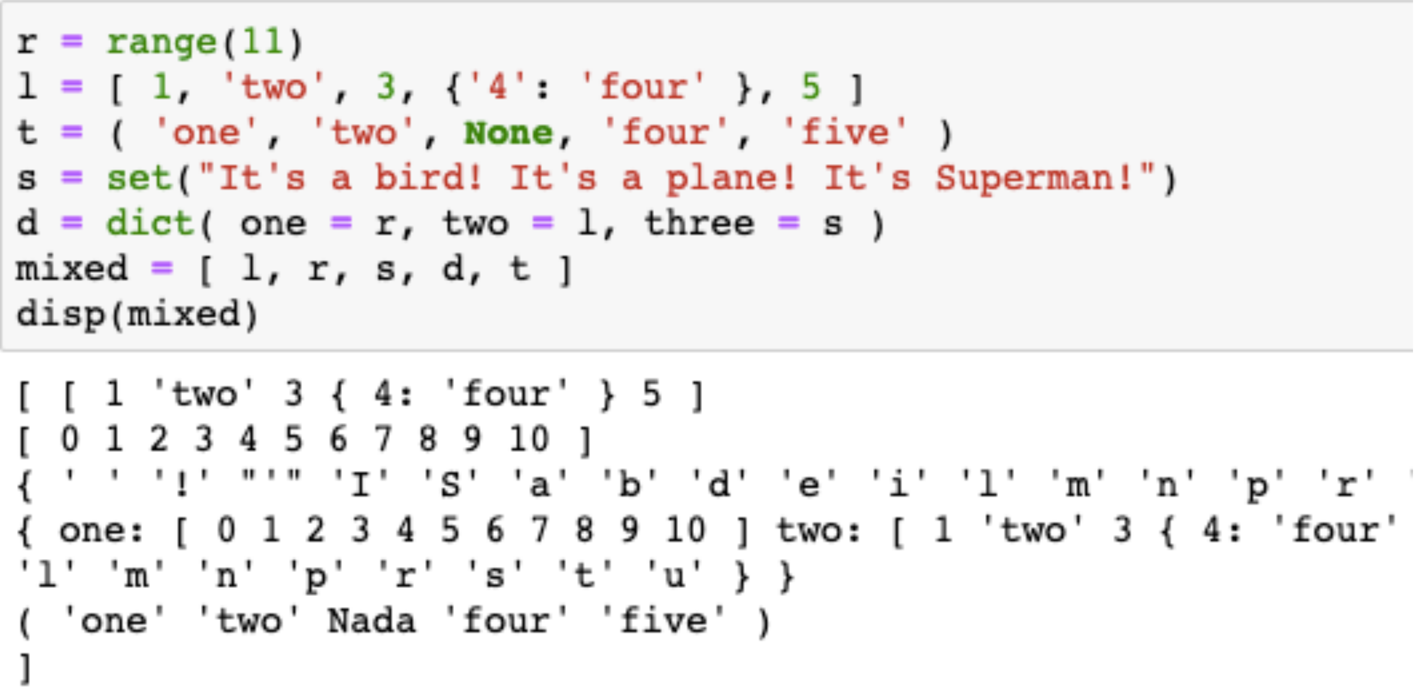
```
[10, 9, 8, 7, 6]
```

# Use lots of comprehensions!

- They are concise and expressive way to write a data transformation
- They are quick to write, easy to parse, and surprisingly powerful

# Mixed structures

- It is possible to store anything in a data structure
- Here, we have an r range, l list including integers, strings, a dictionary etc



```
r = range(11)
l = [ 1, 'two', 3, {'4': 'four' }, 5 ]
t = ( 'one', 'two', None, 'four', 'five' )
s = set("It's a bird! It's a plane! It's Superman!")
d = dict( one = r, two = l, three = s )
mixed = [ l, r, s, d, t ]
disp(mixed)
```

```
[ [ 1 'two' 3 { 4: 'four' } 5 ]
[ 0 1 2 3 4 5 6 7 8 9 10 ]
{ ' ' '!' "'" 'I' 'S' 'a' 'b' 'd' 'e' 'i' 'l' 'm' 'n' 'p' 'r' '
{ one: [ 0 1 2 3 4 5 6 7 8 9 10 ] two: [ 1 'two' 3 { 4: 'four'
'l' 'm' 'n' 'p' 'r' 's' 't' 'u' } }
( 'one' 'two' Nada 'four' 'five' )
]
```