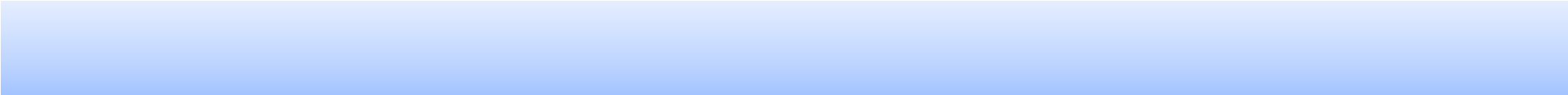


# Syllabus

* Lists
* Dictionaries
* Tuples
* Comprehensions

# Lists

* Lists are Python’s most flexible ordered collection object type.
* Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists.
* Also, unlike strings, lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements and more—they are mutable objects
* Created using square brackets against the object name **L = [‘C’, ‘C++’, ‘Perl’, ‘Python’, ‘Java’]**



# Key Properties of Lists

* Ordered collection of arbitrary objects
* Accessed by offset
* Variable-length, heterogeneous and arbitrarily nestable
* Mutable
* Technically, the list contains zero or more references to other objects

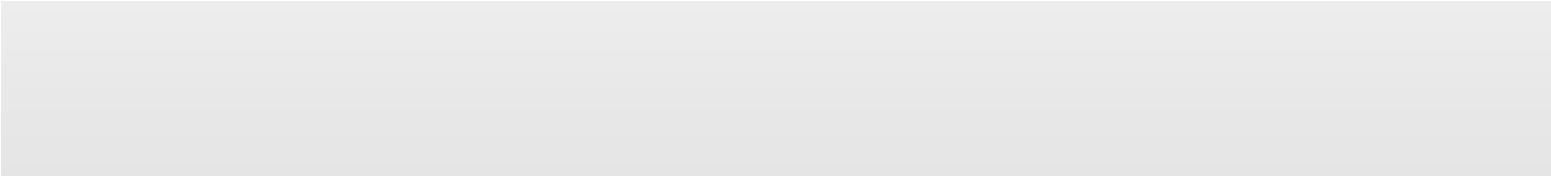
Typical collection/list in python



|  |  |
| --- | --- |
| 0 | "Mindful Learning" |
| 1 | 100 |
| 2 | “www.google.com” |
| 3 | 1.9828 |
| 4 | 0 |
| 5 | [0, 1, 1, 2, 3] |
| 6 | None |
| 7 | 3.33E-01 |

# Accessing Values in Lists

• Values in lists can be access by specifying the index value in square brackets



]

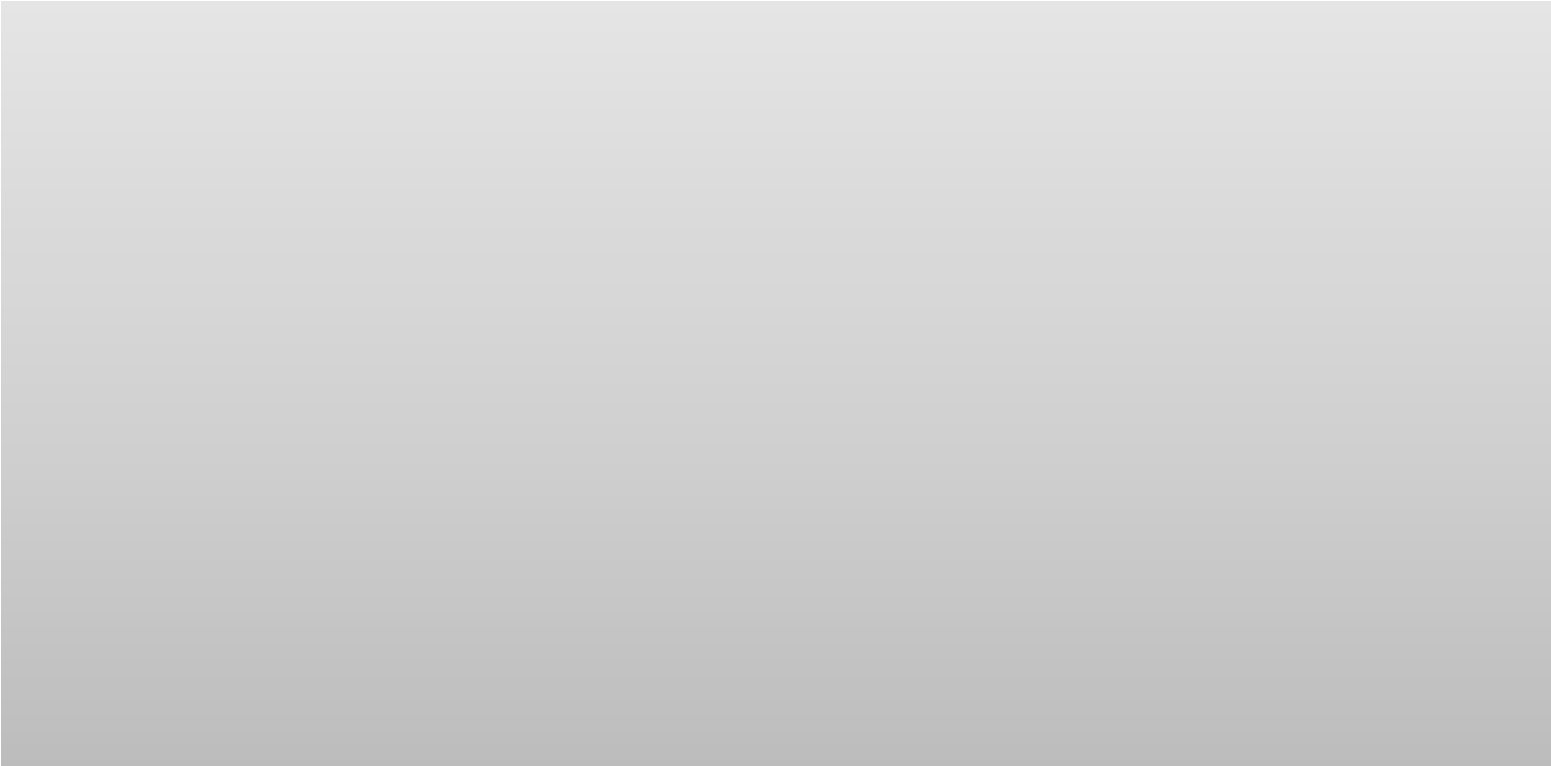
>>>

list1 = ['a', 'e', 'i', 'o', 'u'

>>>

list

1[0]



'a'

>>>

list

1[2:]

[

'i', 'o', 'u'

]

>>>

list

1[::2]

[

'a', 'i', 'u'

]

>>>

list

1[-3]

'i‘

>>>

list

1[-3:-1]

[

'i', 'o'

]

>>>

list1[-1:-3] # Don’t use

[]

>>>

list

1[::-1]

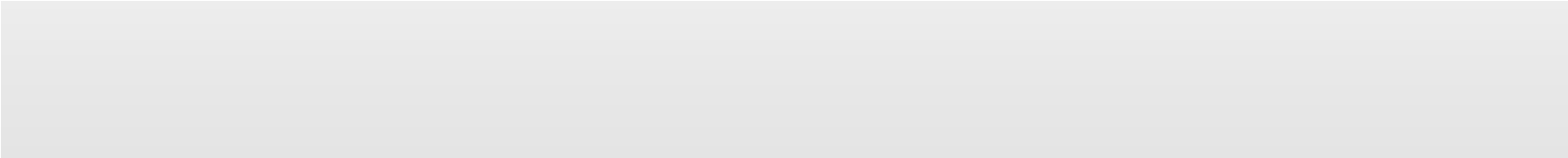
[

'u', 'o', 'i', 'e', 'a'

]

# Updating Lists

• List are mutable and they can be changed in place as shown in the below examples:



>>>

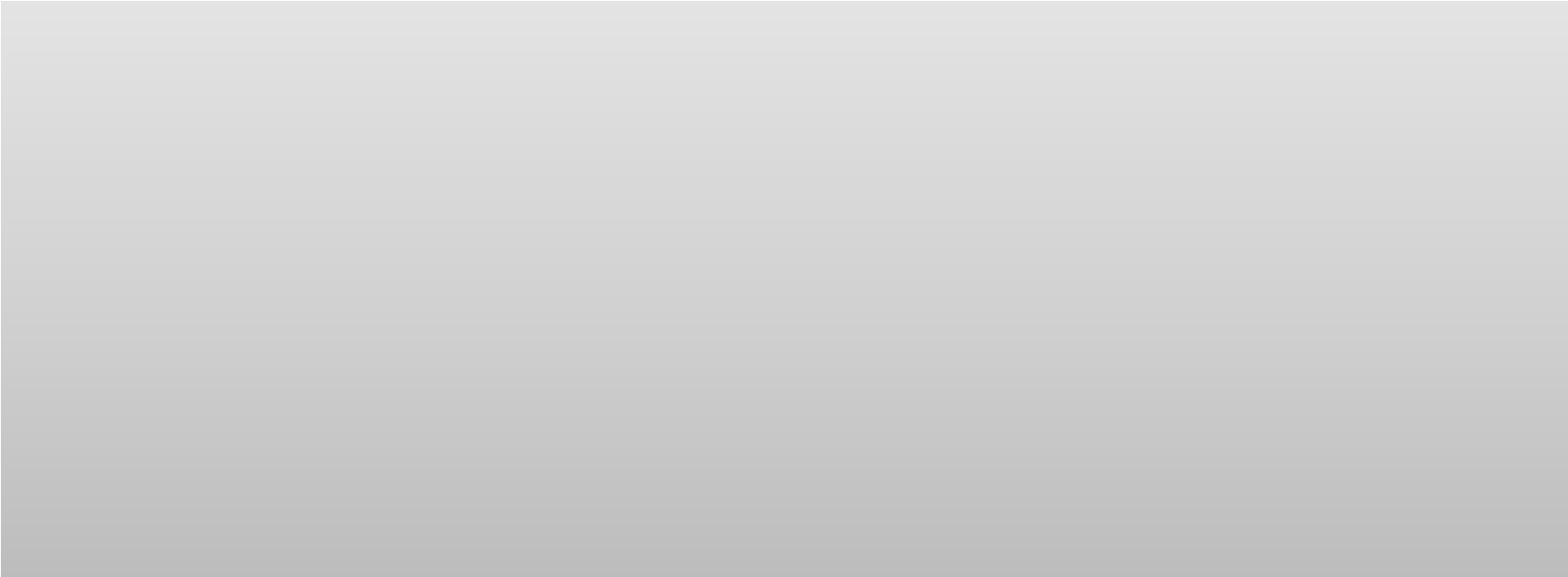
list

1

[

'a', 'e', 'i', 'o', 'u'

]



>>>

list1[1] = 'E'

>>>

list

1

[

'a', 'E', 'i', 'o', 'u'

]

>>>

list1[2:4] = ['I', 'O', 'U'

]

>>>

list

1

[

'a', 'E', 'I', 'O', 'U', 'u'

]

>>>

list1[7] = 'X'

Traceback(most recent call last):

File "<pyshell#16>", line 1, in <module>

list1[7] = 'X'

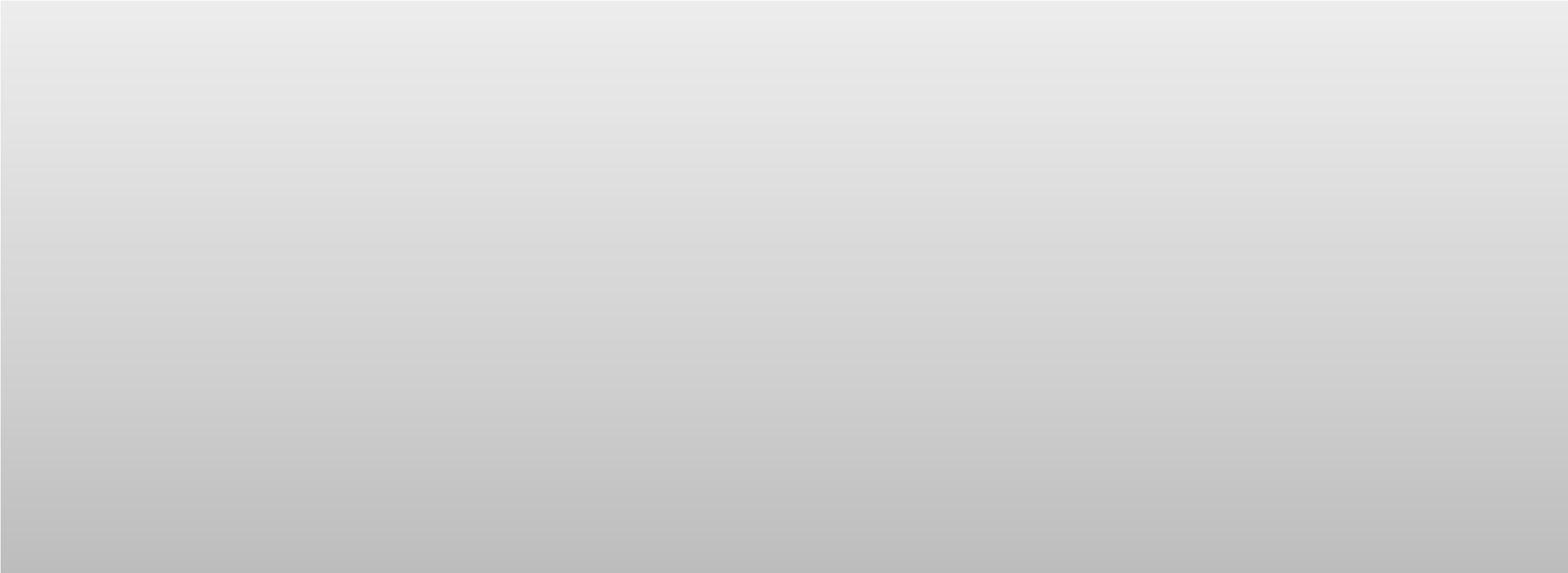
IndexError: list assignment index out of range

# List Methods

|  |  |
| --- | --- |
| Operation | Result |
| del s[i:j] | same as s[i:j] = [] |
| s[i:j:k] = t | the elements of s[i:j:k] are replaced by those of *t* |
| del s[i:j:k] | removes the elements of s[i:j:k] from the list |
| s.append(x) | appends *x* to the end of the sequence (same as s[len(s):len(s)] = [x]) |
| s.clear() | removes all items from s (same as del s[:]) |
| s.copy() | creates a shallow copy of s (same as s[:]) |
| s.extend(t) or **s += t** | extends *s* with the contents of *t* (for the most part the same as s[len(s):len(s)] = t) |
| s \*= n | updates *s* with its contents repeated *n* times |
| s.insert(i, x) | inserts *x* into *s* at the index given by *i* (same as s[i:i] = [x]) |
| s.pop([i]) | retrieves the item at *i* and also removes it from *s* |
| s.remove(x) | remove the first item from *s* where s[i] == x |
| s.reverse() | reverses the items of *s* in place |
| reversed(L) | returns the reversed object, original list is unchanged |

Appending, Inserting and Deleting Elements

* Elements can be appended into the list using **append()**
* Elements can be inserting in a specific position using **insert()**
* Elements can be deleted from the list using **del**



>>>

L = ['Boney M', 'Pink Floyd', 'Scorpions'

]

>>>

L

[

'Boney M', 'Pink Floyd', 'Scorpions'

]

>>>

L.append('Metallica'

)

>>>

L

[

'Boney M', 'Pink Floyd', 'Scorpions', 'Metallica'

]

>>>

del L

[0]

>>>

L

[

'Pink Floyd', 'Scorpions', 'Metallica'

]

>>>

L.insert(1, 'Cold Play'

)

>>>

L.insert(2, 'Collective Soul'

)

>>>

L.insert(3, 'Sting & The Police'

)

>>>

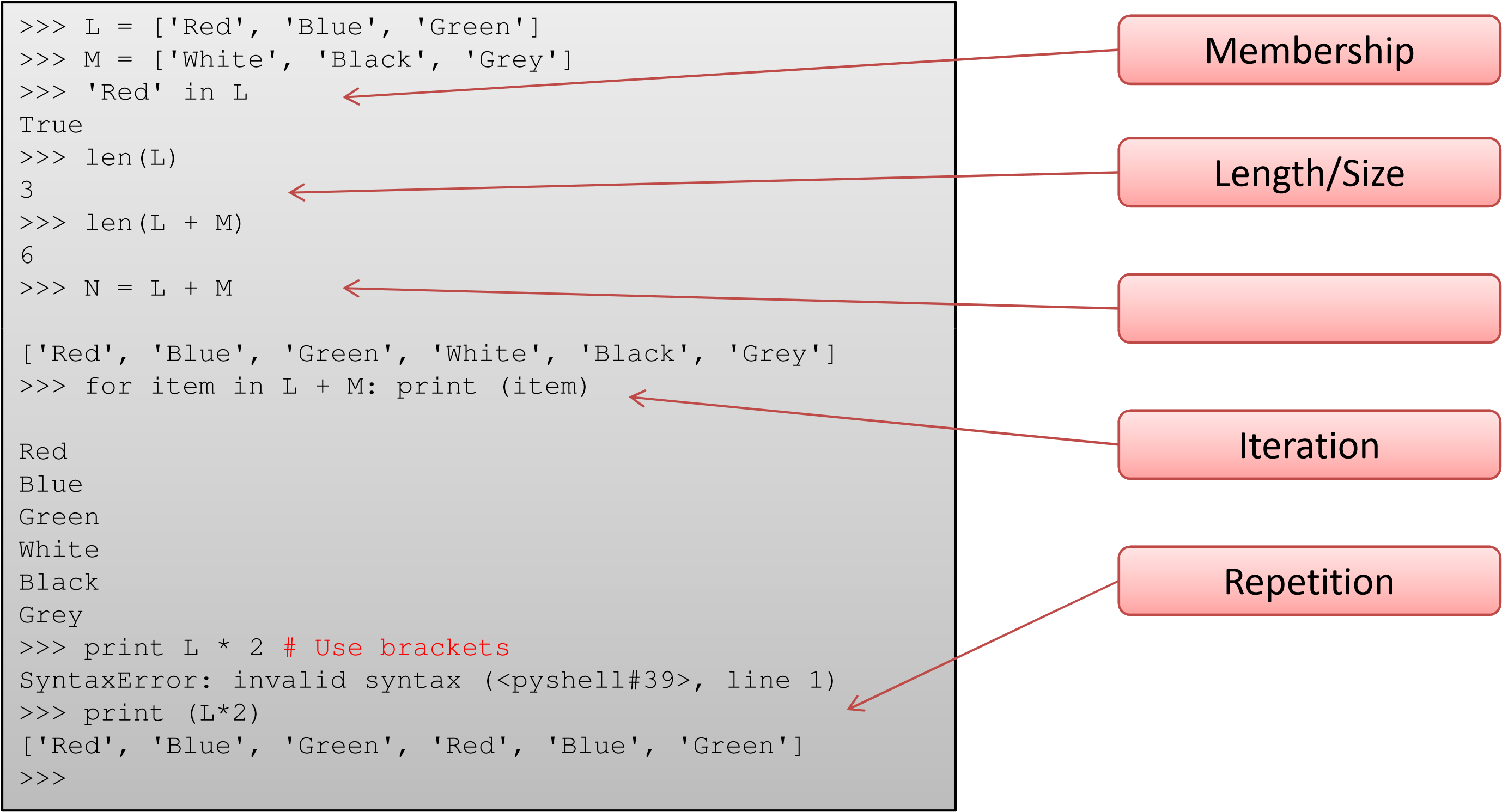
L

[

'Pink Floyd', 'Cold Play', 'Collective Soul', 'Sting & The Police', 'Scorpions',

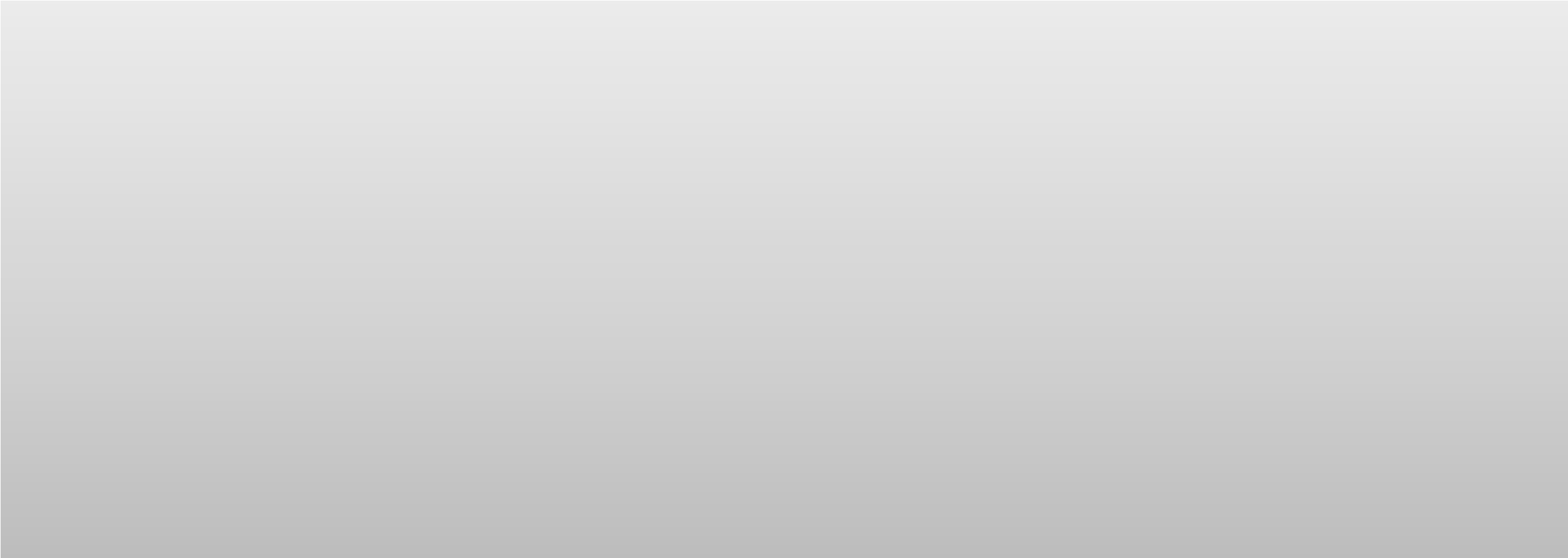
'Metallica']

# Basic List Operations



Minimum, Maximum, Comparison

* **max()** return the item which has the maximum value
* **min()** returns the item which has the minimum value
* **cmp()** compares two lists (Doesn’t work in 3.x)



**Python 3.1.1**

(

r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on

win32

Type "copyright", "credits" or "license()" for more information.

>>>

L = ['Apple', 'Banana', 'Cherry'

]

>>>

M = ['P', 'Q', 'R'

]

>>>

N = [1, 2,

3]

>>>

max(L), max(M), max(N

)

(

'Cherry', 'R',

3)

>>>

max (L+M

)

'R'

>>>

max (M+N

)

Traceback(most recent call last):

File "<pyshell#5>", line 1, in <module>

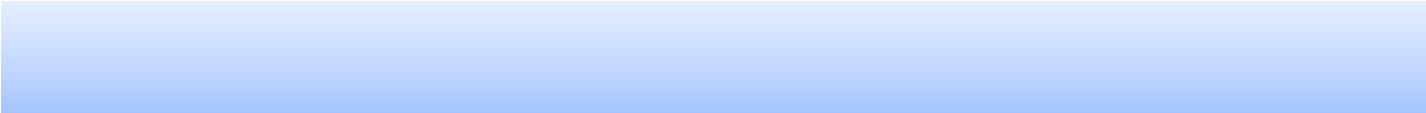
max (M+N)

TypeError: unorderabletypes: int() > str()

# Sorting a List

• Lists can be sorted using the **sort()**

## **sort(\*, key=None, reverse=None)**



* *key* specifies a function of one argument that is used to extract a comparison key from each list element (for example, key=str.lower). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of None means that list items are sorted directly without calculating a separate key value.
* *reverse* is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed
* This method modifies the sequence in place for economy of space when sorting a large sequence

• use **sorted()** to explicitly request a new sorted list instance

# Sorting Examples



>>>

L = ['abc', 'ABD', 'aBe'

]

()

>>>

L.sort

>>>

L

[

'ABD', 'aBe', 'abc'

]

]

>>>

L = ['abc', 'ABD', 'aBe'

>>>

M = sorted(L

)

>>>

M

[

'ABD', 'aBe', 'abc'

]

>>>

L

'abc', 'ABD', 'aBe'

]

[



>>>

L.sort

(

key =

str.lower

)

>>>

L

[

'abc', 'ABD', 'aBe'

]

]

>>>

L = ['abc', 'ABD', 'aBe'

>>>

L.sort(key=str.lower, reverse=True

)

>>>

L

[

'aBe', 'ABD', 'abc'

]

>>>

L.append

(5)

>>>

L

[

'aBe', 'ABD', 'abc',

5]

>>>

L.sort

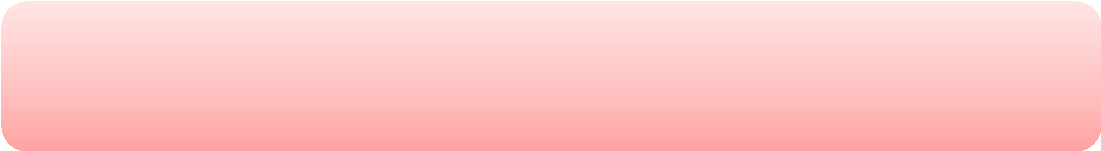
()

Traceback(most recent call last):

File "<pyshell#31>", line 1, in <module>

L.sort()

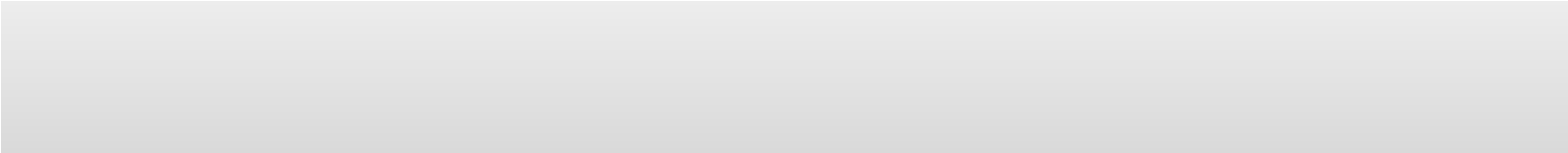
TypeError: unorderabletypes: int() < str()



Sorting of mixed types fails in 3.x

# list()

• The method **list()** takes sequence types and converts them to lists.



>>>

L = list('Jupiter'

)

>>>

L



[

'J', 'u', 'p', '

i

', 't', 'e', 'r']

>>>

L = list(('Io', 'Europa', 'Ganymede', 'Callisto'

))

>>>

L

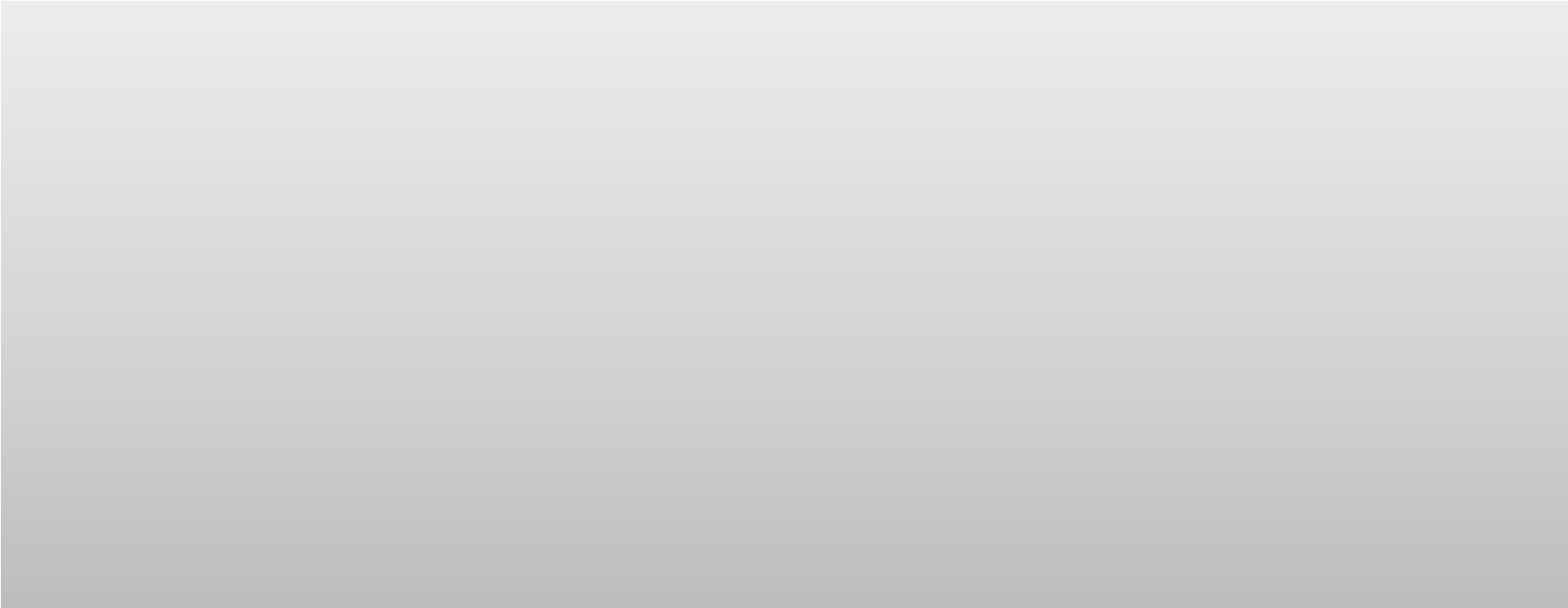
[

'Io', 'Europa', 'Ganymede', 'Callisto'

]

## Two Dimensional Matrices

* One of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sub-lists
* This is also called nesting of lists



>>>

matrix= [[1, 2, 3], [4, 5, 6], [7, 8,

9]]

>>>

matrix

[1]

[4

, 5,

6]

>>>

matrix

[1][2]

6

>>>

matrix[1][0:1] = ['A', 'B'

]

>>>

matrix

[[1

, 2, 3], ['A', 'B', 5, 6], [7, 8,

9]]

>>>

matrix[1].remove('B'

)

>>>

matrix

[[1

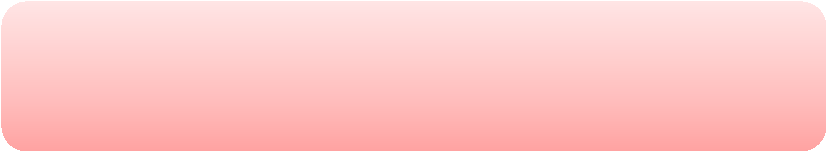
, 2, 3], ['A', 5, 6], [7, 8,

9]]

## A Practical Application

* Application: General
* Given a list, remove all the duplicate elements from the list

Time: 15 min + 10 min



## Solution #1

>>> t = [1, 2, 3, 1, 2, 5, 6, 7, 8] >>> for i in t:

... if i in t[t.index(i)+1:]: ... t.remove(i) ...

>>> t

[3, 1, 2, 5, 6, 7, 8]

def remove\_duplicates(list):

''' Removes duplicate items from a list ''' singles\_list = [] for element in list:

if element not in singles\_list:

singles\_list.append(element)

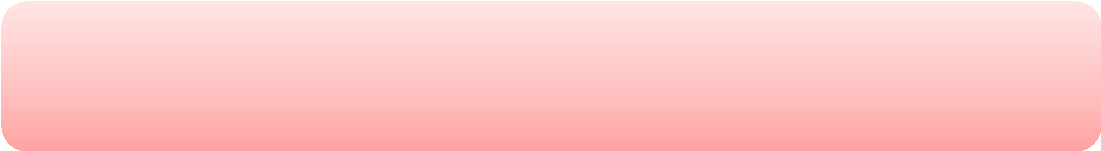
return singles\_list

a = [1,2,3,4,5,9,11,15] b = [4,5,6,7,8] c=a+b print c print list(set(c)) #one line for getting unique elements of c

## A Practical Application

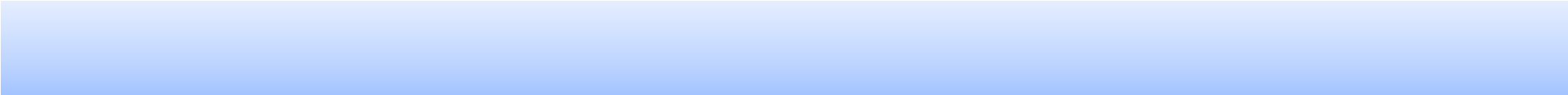
* Application: Matrices
* Write a python script to add and multiply two matrices

Time: 20 min + 20 min + 10 min



## Dictionaries

* Along with lists, dictionaries are one of the most flexible builtin data types in Python.
* If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections;
* The chief distinction is that in dictionaries, items are stored and fetched by key, instead of by positional offset.
* While lists can serve roles similar to arrays in other languages, dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions
* Created using flower brackets against the object name **D = {‘Name’ : ‘Mark’, ‘EID’ : 123456}**

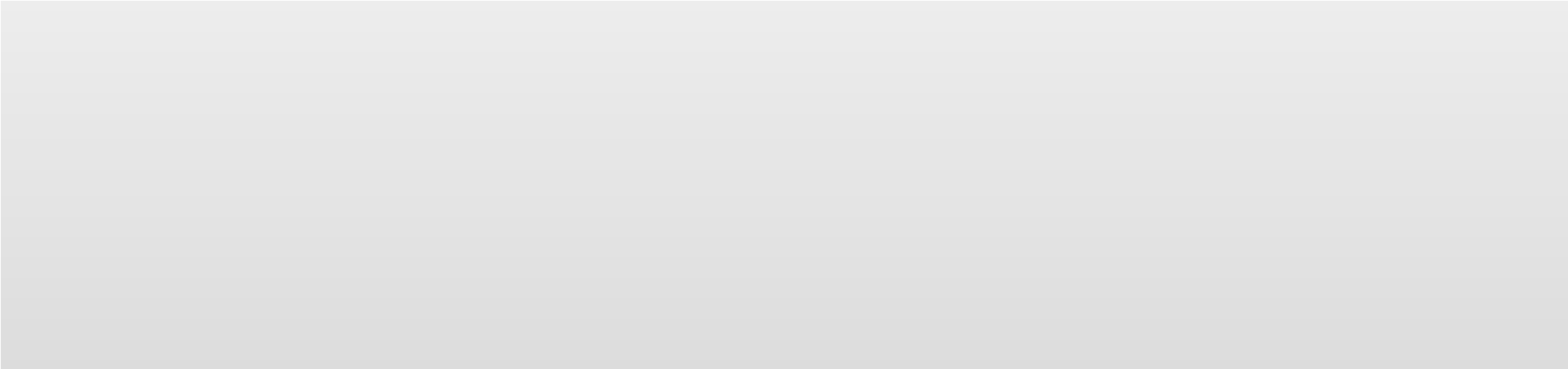


## Key Features of Dictionaries

* Accessed by key, not by offset position
* Un-ordered collection of arbitrary objects
* Variable length, heterogeneous and arbitrarily nestable
* Mutable
* They are tables of object references (hash tables)

|  |  |
| --- | --- |
| **Key** | **Value** |
| ‘Alexandre Dumas’ | ‘The count of Monte Cristo’ |
| ‘Sidney Sheldon’ | ‘Master of the Game’ |
| ‘John Bunyan’ | ‘Pilgrim’s Progress’ |
| ‘Mark Twain’ | ‘Huckleberry Finn’ |
| ‘Ian Flemming’ | ‘Casino Royale’ |
| ‘JRR Tolkein’ | ‘The Lord of the Rings’ |
| ‘Ayan Rand’ | ‘Atlas Shrugged’ |
| ‘Sir Arthur Conan Doyle’ | ‘The Hound of the Baskervilles‘ |

## Accessing Dictionaries



>>>

D = { 'Author' : 'Mark Lutz',

'Title': 'Learning Python',

'Publisher' : 'Shroff; FifthEdition (2013)',

'Language' : 'English',

'ISBN-10' : '9351102017',

'ISBN-13': '978-9351102014',

'AverageReview': '4/5',

'Amazon Best Seller\'s Rank': 24774 }

>>>

D

**# Ordergetsscrambled**



Creating a dictionary



{

"Amazon Best

Seller's

Rank": 24774, 'Publisher': '

Shroff

;

Fifth

Edition (2013)',

'ISBN-10': '9351102017', 'Language': 'English', 'Author': 'Mark Lutz', 'ISBN-13':

'978-9351102014', 'Title': 'Learning Python', 'AverageReview': '4/5'}

>>>

D['Publisher']

**# Access by keyname**

'Shroff; FifthEdition (2013)'

>>>

type (D

)

<

class 'dict'

>

>>>

len(D)

**# Returnsthe size of the dictionary**

8

>>>

'English' in D

**# use in withkeysnot values**

False

>>>

'Language' in D

True

>>>

list(D.keys())

**# Returnsa listof keys**

[

"Amazon Best Seller'sRank", 'Publisher', 'ISBN-10', 'Language', 'Author', 'ISBN

-

13

', 'Title', 'AverageReview'

]

## Updating Dictionaries

>>> str(D) **# Converts D into a string**

"{'Publisher': 'Shroff; Fifth Edition (2013)', 'Paperback': '1594 Pages', 'ISBN10': '9351102017', 'Language': 'English', 'Author': 'Lutz, Mark', 'ISBN-13': '978-

9351102014', 'Title': 'Learning Python', 'Average Review': '4/5'}"

>>> D['Author'] = 'Lutz, Mark‘ **# Changing in place**

>>> D['Author']

'Lutz, Mark'

>>> D['Paperback'] = '1594 Pages‘ **# Adding new element**

>>> D

{"Amazon Best Seller's Rank": 24774, 'Publisher': 'Shroff; Fifth Edition (2013)',

'Paperback': '1594 Pages', 'ISBN-10': '9351102017', 'Language': 'English',

'Author': 'Lutz, Mark', 'ISBN-13': '978-9351102014', 'Title': 'Learning Python',

'Average Review': '4/5'}

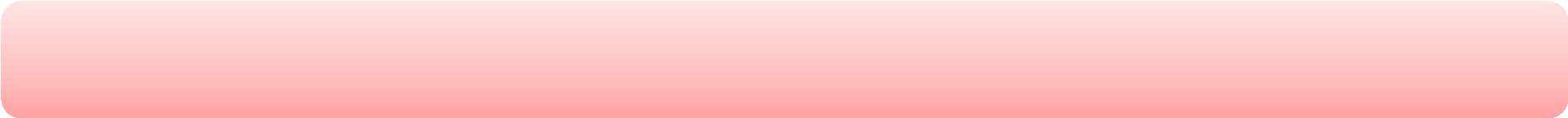
>>> del D['Amazon Best Seller\'s Rank'] **# Deleting an element**

>>> D

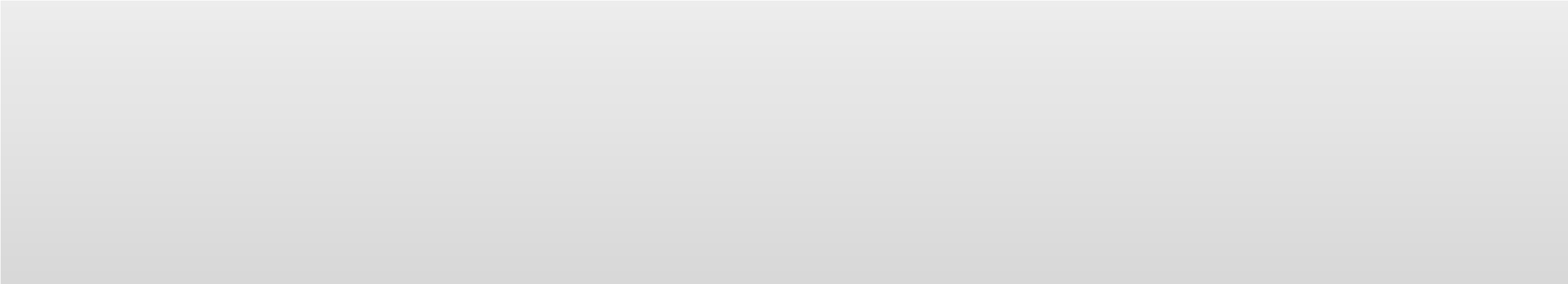
{'Publisher': 'Shroff; Fifth Edition (2013)', 'Paperback': '1594 Pages', 'ISBN10': '9351102017', 'Language': 'English', 'Author': 'Lutz, Mark', 'ISBN-13': '9789351102014', 'Title': 'Learning Python', 'Average Review': '4/5‘}



**cmp(D1, D2)** can be used to compare dictionaries



## Updating Dictionaries



>>>

list(D.items())

**# Returnsa listof tuples**

[(

'Publisher', 'Shroff; FifthEdition (2013)'), ('Paperback', '1594 Pages'), ('ISBN

-

10

', '9351102017'), ('Language', 'English'), ('Author', 'Lutz, Mark'), ('ISBN-13',

'978-9351102014'), ('Title', 'Learning Python'), ('AverageReview', '4/5')]

>>>

for (item\_key, item\_value) in list(D.items()):

**# Iteratingthrougha dictionary**

print(item\_key, item\_value)



Publisher Shroff; FifthEdition (2013)

Paperback1594 Pages

ISBN-10 9351102017

LanguageEnglish

AuthorLutz, Mark

ISBN-13 978-9351102014

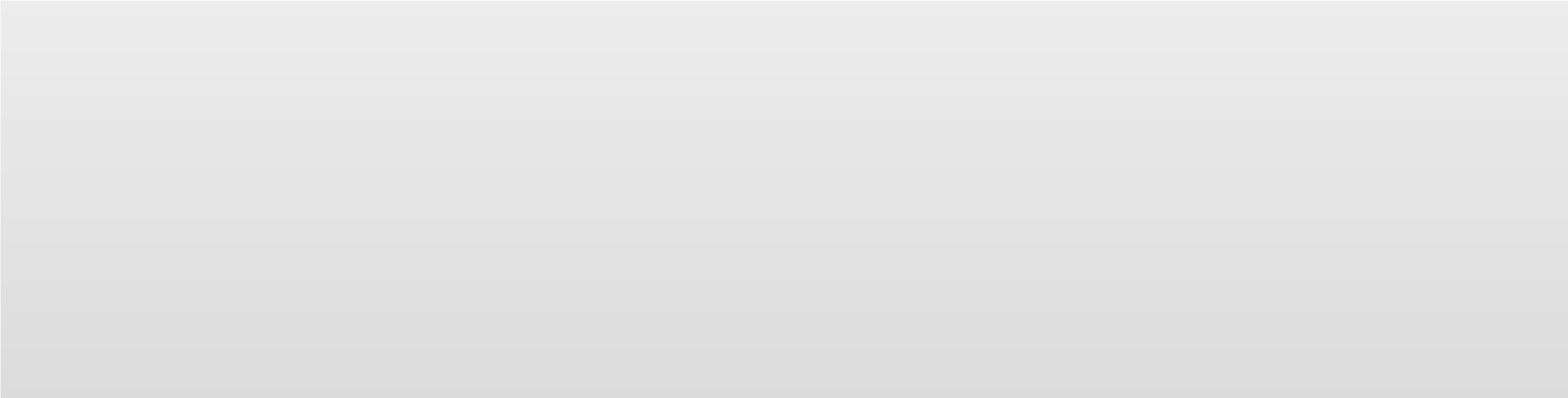
TitleLearning Python

AverageReview4/5

## Dictionary Methods

|  |  |
| --- | --- |
| **Methods** | **Interpretation** |
| D.keys() | Get all keys |
| D.values() | Get all values |
| D.items() | All key-value tuples |
| D.copy() | Copy (top-level) |
| D.clear() | Remove all items |
| D.update(D2) | Merge by keys |
| D.get(key, default?) | Fetch by key; if absent default(or None) |
| D.pop(key, default?) | Remove by key; if absent default(or Error) |
| D.setdefault(key, default?) | Fetch by key; if absent set default(or None) |
| D.popitem() | Remove/return any key-value pair |
| D1.keys() & D2.keys() | Dictionary views |

## Dictionary Methods Examples



>>>

D2 = D.copy()

**# Copyinga dictionary**

>>>

D

2

{

'Publisher': 'Shroff; FifthEdition (2013)', 'Paperback': '1594 Pages', 'Average

Review': '4/5', 'Language': 'English', 'Author': 'Lutz, Mark', 'Title': 'Learning

Python', 'ISBN-10': '9351102017', 'ISBN-13': '978-9351102014'}

>>>

D2.keys

()

dict\_keys(['Publisher', 'Paperback', 'AverageReview', 'Language', 'Author', 'Title',

'ISBN-10', 'ISBN-13'])

>>>

D2.values

()

dict\_values(['Shroff; FifthEdition (2013)', '1594 Pages', '4/5', 'English', 'Lutz,

Mark', 'Learning Python', '9351102017', '978-9351102014'])



>>>

D2.get('Rank'

)

)

>>>

D2.get('Language'

'English'

>>>

D2.pop('Rank'

)

Traceback(mostrecentcall last):

File "<pyshell#35>", line 1, in <module>

D2.pop('Rank')

KeyError: 'Rank'

>>>

D2.pop('Language'

)

'English'

>>>

D2.get('Language'

)

()

>>>

D.keys() & D2.keys

{

'Publisher', 'Paperback', 'AverageReview', 'Title', 'Author', 'ISBN-10', 'ISBN-13'

}

>>>

D2.setdefault('Rank'

)

>>>

D

2

{

'Publisher': 'Shroff; FifthEdition (2013)', 'Paperback': '1594 Pages', 'Average

Review': '4/5', 'Author': 'Lutz, Mark', 'Title': 'Learning Python',

**'Rank': None**

, 'ISBN-

10

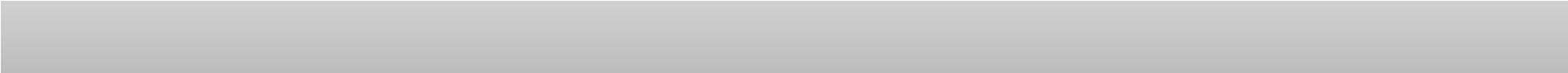
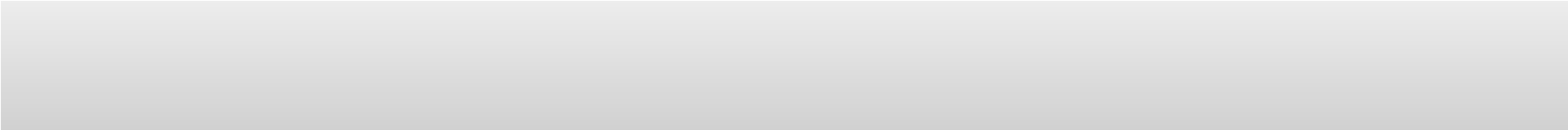
': '9351102017', 'ISBN-13': '978-9351102014‘

}

## Nested Dictionaries

• Dictionaries in python can be nested and can be accessed in the following way:

>>> D = {'CTO': {'First Name': 'Scott', 'Last Name' : 'Baynes', 'Company' : 'Netgain'}, 'EID' : 'NG001' }

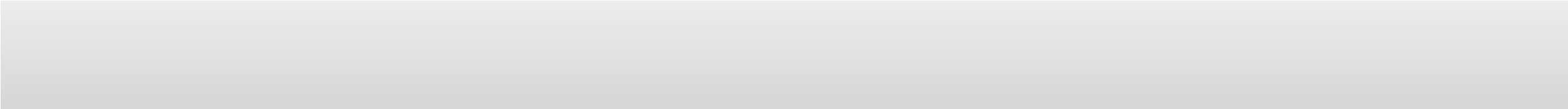


>>> D['CTO']['First Name']

'Scott'

# dict()

• The **dict()** method builds dictionaries directly from sequences of key-value pairs



>>>

T = (('First Name', 'James'), ('Last Name', 'Bond'), ('Code', '007'),

(

'Rank', 'Commander'

))



>>>

D =

dict

(

T

)

>>>

D

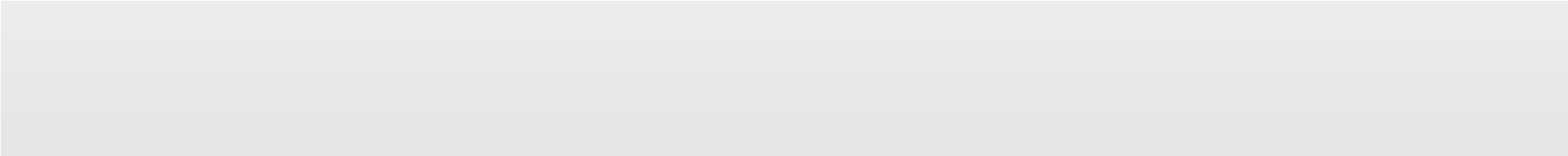
{

'Last Name': 'Bond', 'First Name': 'James', 'Code': '007', 'Rank': 'Commander'

}

## Sorting Dictionaries

• Dictionaries should be sorted with respect to their key values



>>>

D = {1: 'D', 2: 'B', 4: 'E', 5: 'A', 3: 'B', 6 : 'M'

}

>>>

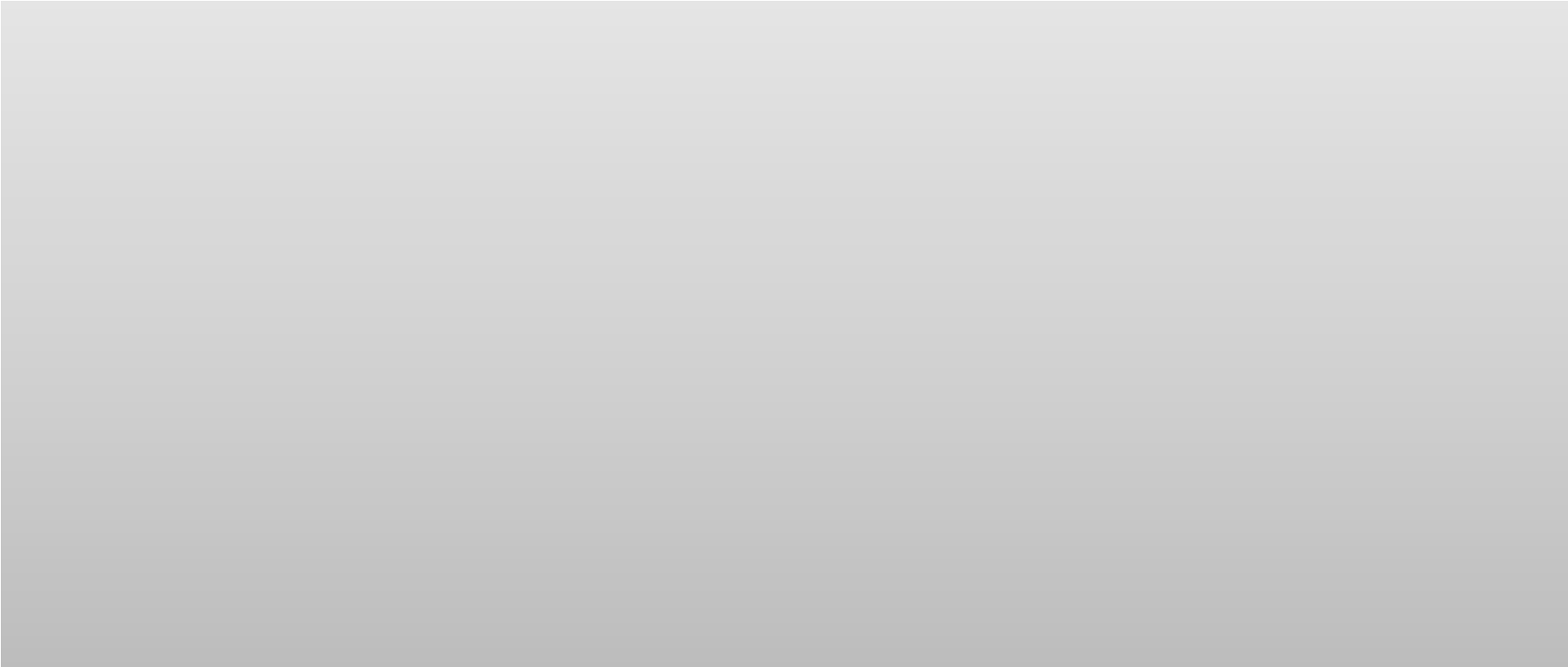
sorted(D

)

[1

, 2, 3, 4, 5,

6]



>>>

sort(D

)

Traceback(most recent call last):

File "<pyshell#36>", line 1, in <module>

sort(D)

NameError: name 'sort' is not defined

>>>

D.sort

()

Traceback(most recent call last):

File "<pyshell#37>", line 1, in <module>

D.sort()

AttributeError: 'dict' object has no attribute 'sort‘

>>>

D = dict(name = 'SatyaNadella', job = 'CEO', company =

'Microsoft')

>>>

sorted(D, reverse=True

)

[

'name', 'job', 'company'

]

## Dictionary Usage

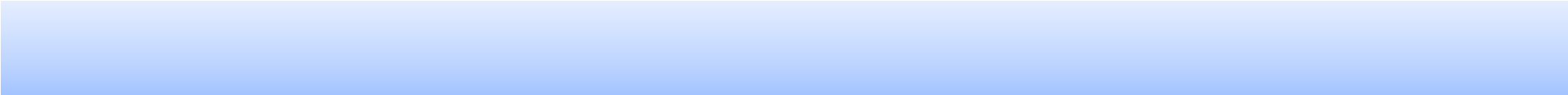
• Some pointers and reminders to be aware of:

* Sequence operations don’t work: Dictionaries are mappings, not sequences; because there’s no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting a contiguous section) simply don’t apply
* Assigning to new indexes adds entries
* Keys need not always be strings

## Tuples

* Tuples construct simple groups of objects.
* They work exactly like lists, except that tuples can’t be changed in place (they’re immutable) and are usually written as a series of items in parentheses, not square brackets.
* Although they don’t support as many methods, tuples share most of their properties with lists.

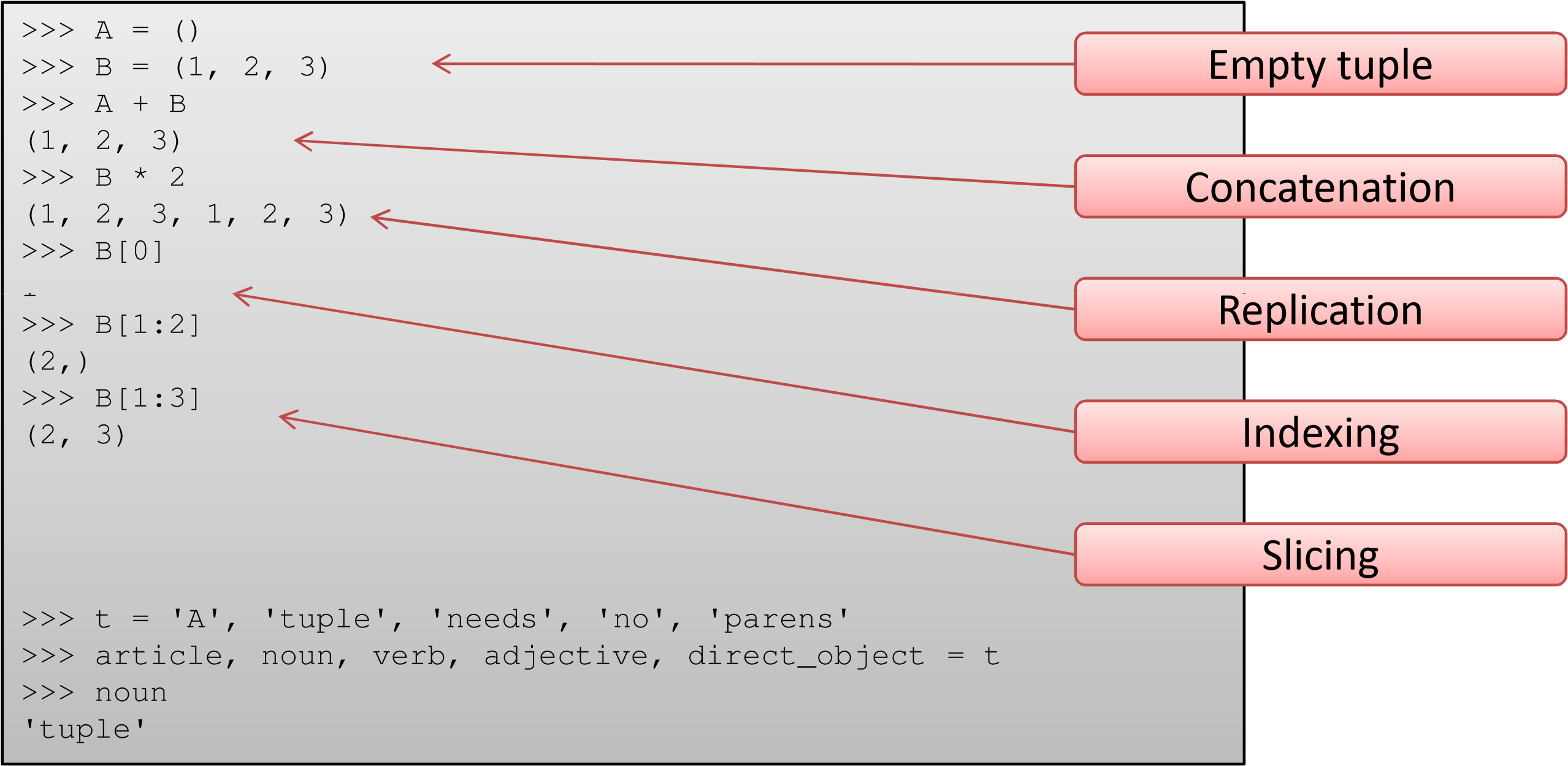
**T = (‘C’, ‘C++’, ‘Perl’, ‘Python’, ‘Java’)**



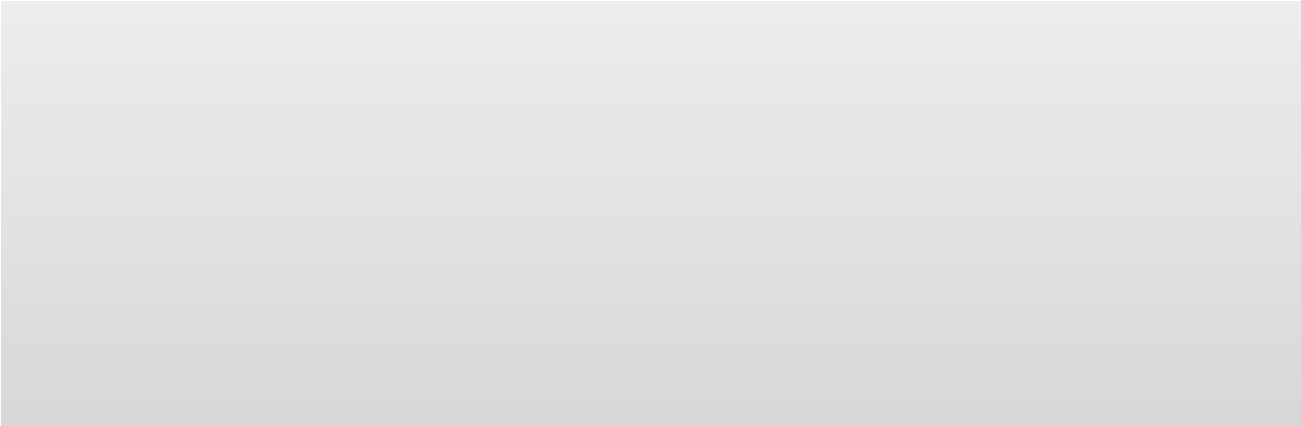
## Key Properties of Tuples

* Ordered collection on arbitrary objects
* Accessed by offset
* Immutable sequence
* Fixed length, heterogeneous and arbitrarily nestable
* Arrays of object references
* Tuples can be used in place of lists where the number of items is known and small, for example when returning multiple values from a function

## Accessing Tuples



## Conversions and Methods



)

temp = list(T

>>>

()

temp.sort

>>>

>>>

temp

[

'aa', 'cc', 'dd', 'gg'

]

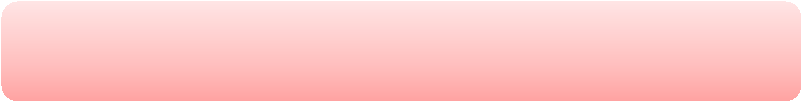
)

>>>

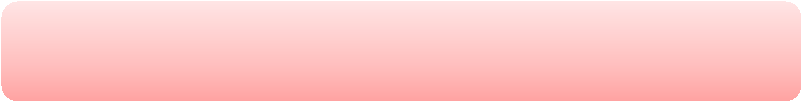
T = tuple(temp

>>>

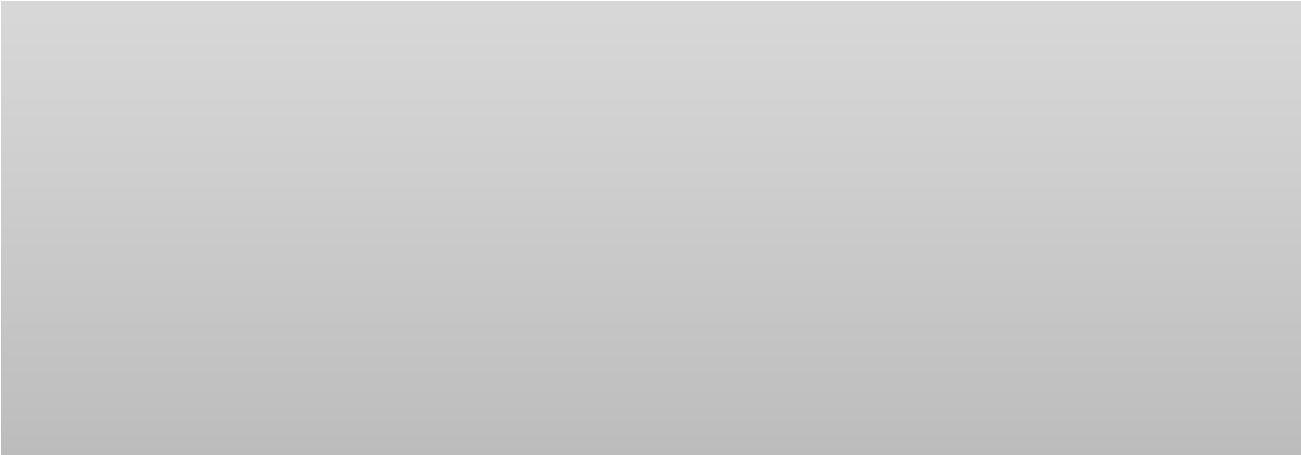
T



Sorting



Count how many



(

'

aa

', 'cc', '

dd

', '

gg

')

>>>

)

T = ('cc', 'aa', 'dd', 'gg'

)

sorted(T

>>>

[

'aa', 'cc', 'dd', 'gg'

]

>>>

T.count('aa'

)

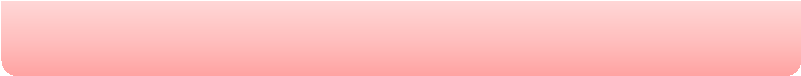
1

>>>

T.index('gg'

)

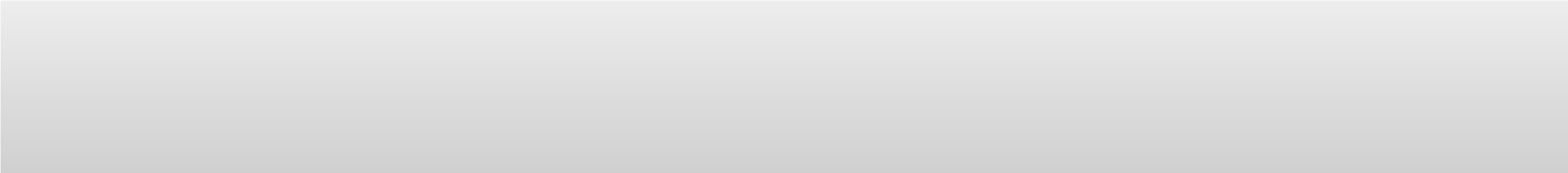
3



Position

## Nesting

• Other types can be nested inside a tuple



>>>

T = ('Red', ['Sky Blue', 'Navy Blue'], ('yellow', 'golden', 'orange'

))

>>>

T2 = ({'Sunglass' : ['Wayfarer', 'Aviator'] }, {'Shoes' : ['Sneakers', 'Slip-ons'

]})

>>>

T2[0]['Sunglass'

]

[

'Wayfarer', 'Aviator'

]



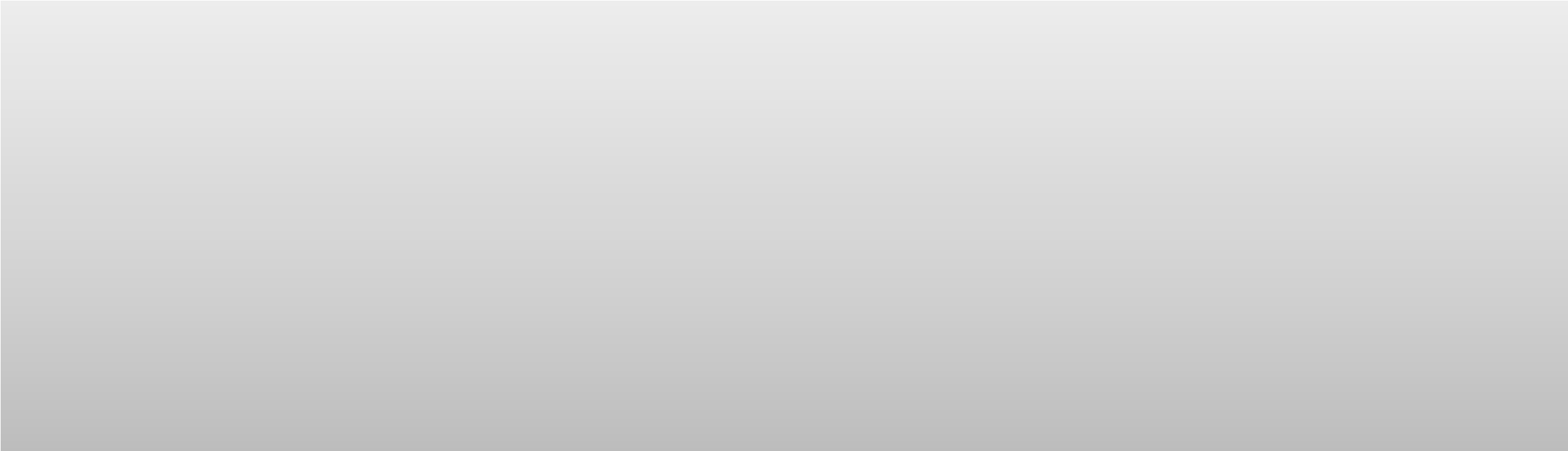
>>> T[2][1]

'golden'

>>>

# itemgetter()

* The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python
* **itemgetter()** is used to retrieve elements
  + Return a callable object that fetches *item* from its operand using the operand’s **\_\_getitem\_\_()** method
  + If multiple items are specified, returns a tuple of lookup values.
  + After f = itemgetter(2), the call f(r) returns r[2].
  + After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])



>>>

from operator import itemgetter

>>>

itemgetter(1)('ABCDEFG'

)

'B'

)

>>>

itemgetter(1,3,5)('ABCDEFG'

(

'B', 'D', 'F'

)

>>>

itemgetter(slice(2,None))('ABCDEFG'

)

'CDEFG'

>>>

inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange',

1)]

>>>

getcount= itemgetter

(1)

>>>

getcount

<

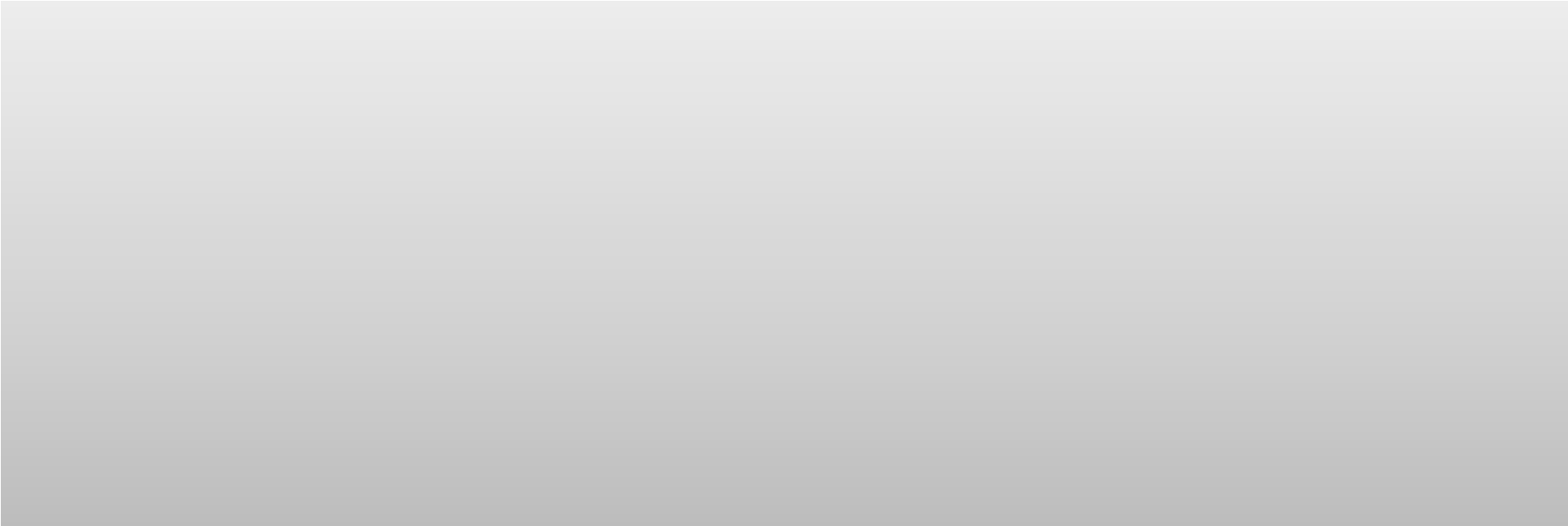
operator.itemgetterobject at

0x0138B250>



Returns a callable getcount()

# itemgetter()



getcount(inventory

>>>

)

2)

'banana',

(

)

map(getcount, inventory

>>>

0>

map object at 0x0138B4B

<

>>>

list(map(getcount, inventory

))

1]

, 2, 5,

[3

)

sorted(inventory, key = getcount

>>>

'orange', 1), ('banana', 2), ('apple', 3), ('pear',

[(

5)]

>>>

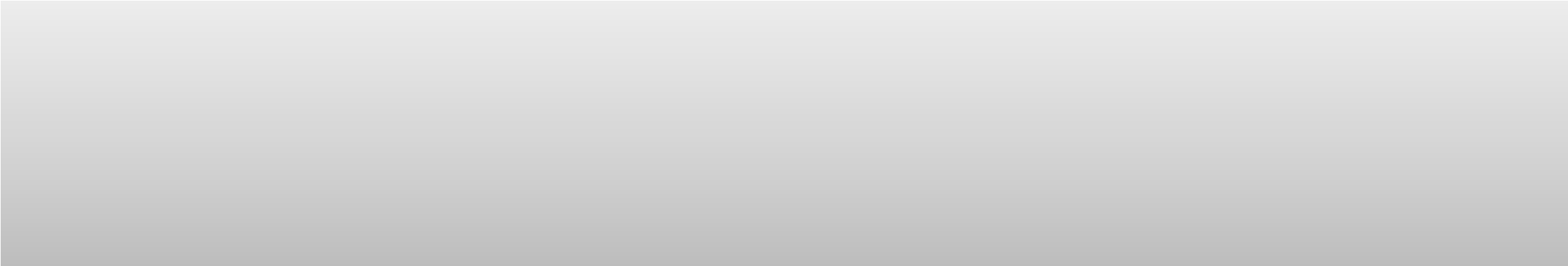
sorted(inventory, key = itemgetter

(1))

[(

'orange', 1), ('banana', 2), ('apple', 3), ('pear',

5)]



1)]

>>>

inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange',

>>>

sorted(inventory, key = itemgetter

(2))

Traceback(most recent call last):

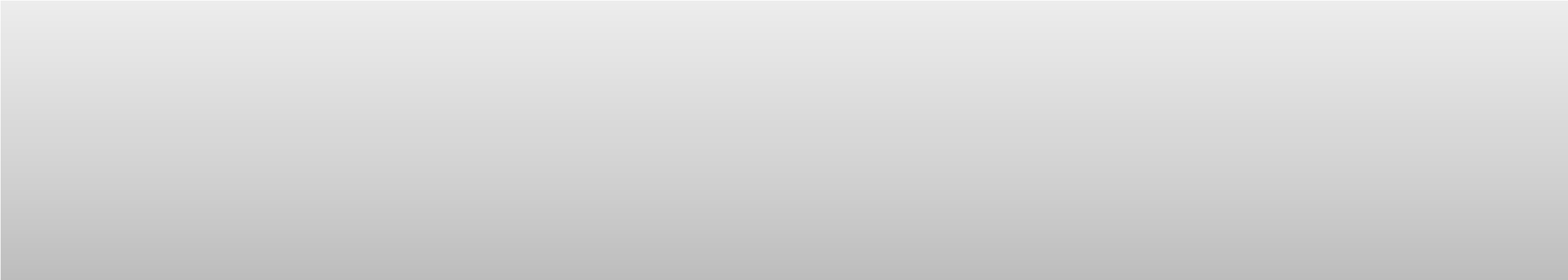
File "<pyshell#32>", line 1, in <module>

sorted(inventory, key = getcount)

IndexError: tupleindex out of range



Why?



>>>

inventory = [('apple', 3, 4.5), ('banana', 2, 6.7), ('pear', 5, 12.4),

(

'orange', 1,

1.2)]

>>>

get = itemgetter

(3)

>>>

get(inventory

)

(

'orange', 1,

1.2)

>>>

sorted(inventory, key = itemgetter

(2))

[(

'orange', 1, 1.2), ('apple', 3, 4.5), ('banana', 2, 6.7), ('pear',

5, 12.4)]

## Sorting Using **itemgetter()**



# initialize

a = []

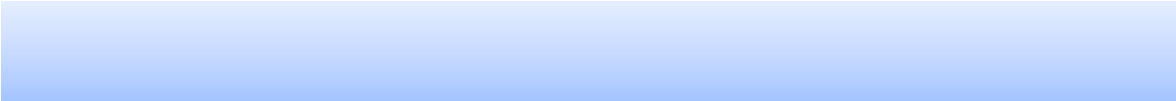
# create the table (name, age, job)

a.append(["Nick", 30, "Doctor"])

a.append(["John", 8, "Student"])

a.append(["Paul", 22, "Car Dealer"])

a.append(["Mark", 66, "Retired"])



a.sort(key=lambda x: x[1])



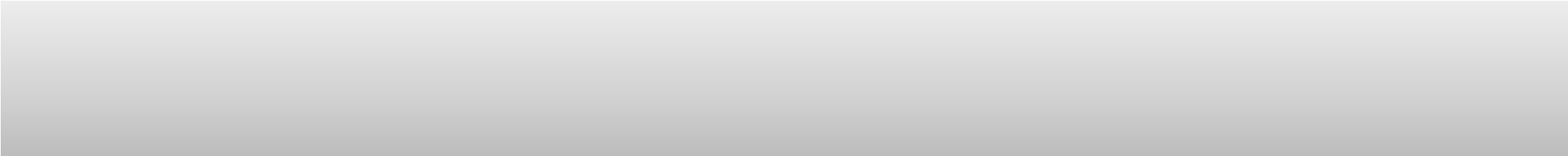
# sort the table by age

import operator

a.sort(key=operator.itemgetter(1))

# print the table

print(a)

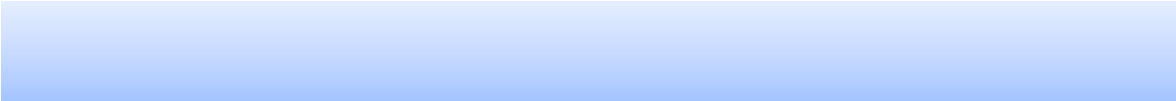


>>>

[[

'John', 8, 'Student'], ['Paul', 22, 'Car Dealer'], ['Nick', 30,

'Doctor'], ['Mark', 66, 'Retired']]



a.sort(key=get\_element(1))

Try sorting with respect to names

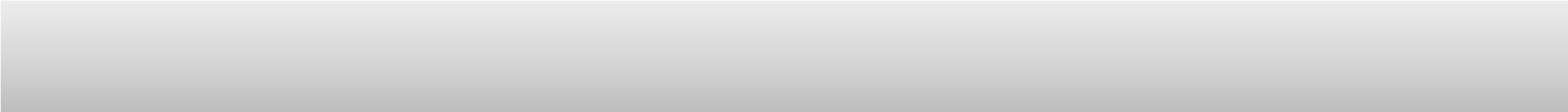


# attrgetter()

* Say, you have a list of ‘Country’ objects. In the Country class there is an attribute called “population“ and you like to sort the object with respect to that attribute:

import operator

countries.sort(key=operator.attrgetter("population"), reverse=False)



* **attrgetter()** retrurns a callable object that fetches *attr* from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:
  + After f = attrgetter('name'), the call f(b) returns b.name.
  + After f = attrgetter('name', 'date'), the call f(b) returns (b.name, b.date).
  + After f = attrgetter('name.first', 'name.last'), the call f(b) returns (b.name.first, b.name.last).

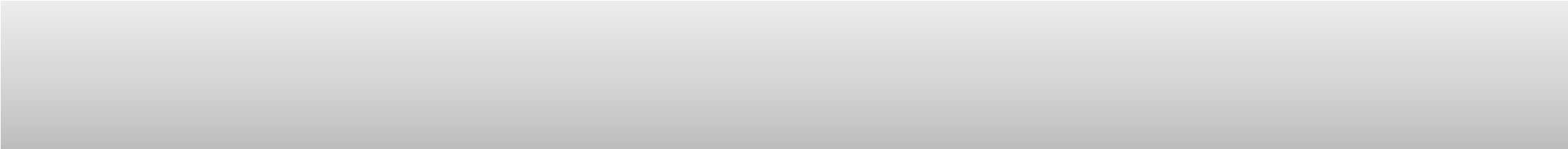
# methodcaller()

* Say, you have a class ‘Student’ and the grades are stored in a list. You want to sort the students by the average of their grades. This value is calculated by the function get\_avg\_grade()

import operator

# students is a list of Student objects

students.sort(key=operator.methodcaller("get\_avg\_grade"), reverse=False)



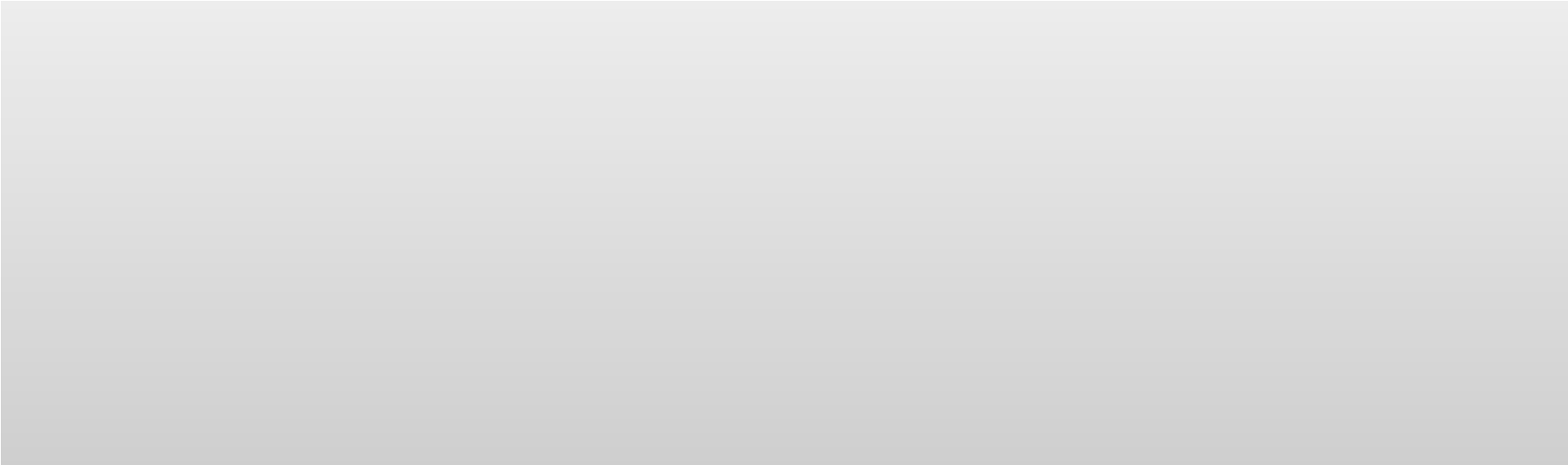
* **methodcaller()** returns a callable object that calls the method *name* on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:
  + After f = methodcaller('name'), the call f(b) returns b.name().
  + After f = methodcaller('name', 'foo', bar=1), the call f(b) returns

b.name('foo', bar=1)

## List Comprehensions

* List comprehension is an elegant way to define and create list in Python
* Common applications are to make lists where each element is the result of some operations applied to each member of the sequence, or to create a subsequence of those elements that satisfy a certain condition.
* A list comprehension consists of brackets containing an expression followed by a **for** clause, then zero or more **for** or **if** clauses.
* The result will be a list resulting from evaluating the expression in the context of the **for** and **if** clauses which follow it.
* If the expression would evaluate to a tuple, it must be parenthesized.

## List Comprehensions: Examples



>>>

vec= [2, 4,

6]

new\_vec= [3\*x for x in vec

]

>>>

>>>

new\_vec

[6

, 12,

18]

>>> [[

x, x\*\*2] for x in vec

]

36]]

[[2

, 4], [4, 16], [6,

3]

>>> [3

\*x for x in vecif x >

[12

,

18]



>>> [

x, x\*\*2 for x in vec]

**#Paranthesisrequired for tuples**

SyntaxError: invalid syntax (<pyshell#52>, line 1)

>>> [(

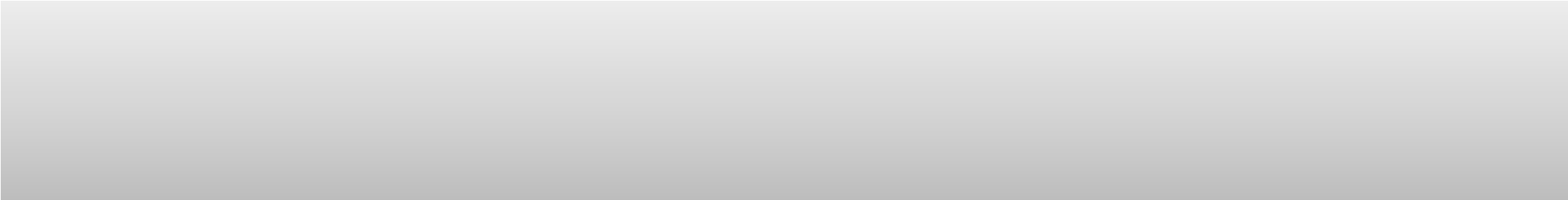
x, x\*\*2) for x in vec

]

[(2

, 4), (4, 16), (6,

36)]



>>>

freshfruit= [' banana', ' loganberry ', 'passion fruit '

]

>>>

freshfruit= [fruit.strip() for fruit in freshfruit

]

>>>

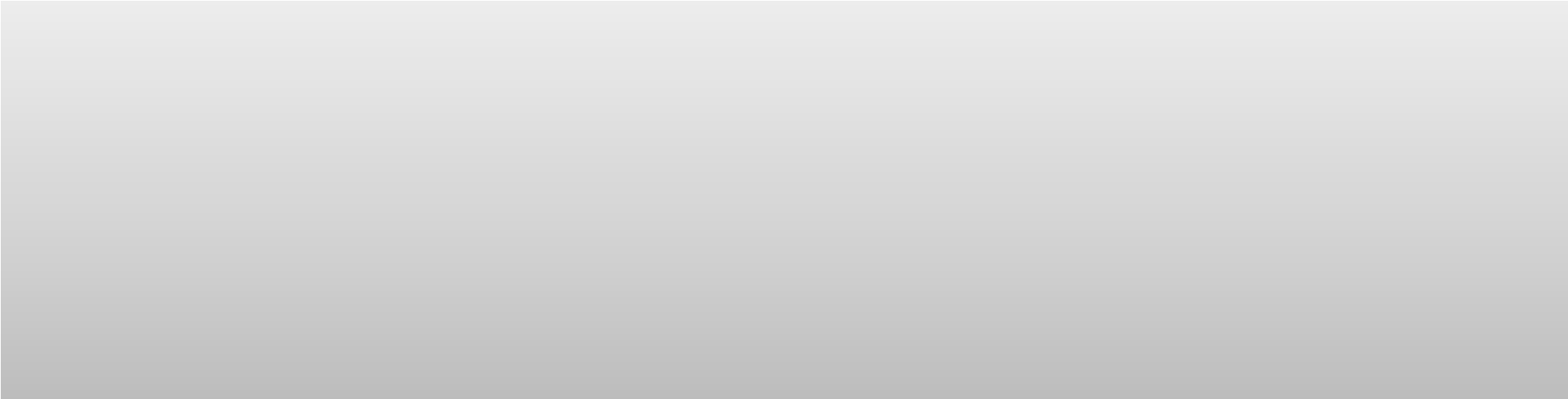
freshfruit

[

'banana', 'loganberry', 'passion fruit'

]

## List Comprehension: Examples



6]

vec1 = [2, 4,

>>>

>>>

-9]

vec2 = [4, 3,

2]

>>> [

x\*y for x in vec1 for y in vec

-54]

, 6, -18, 16, 12, -36, 24, 18,

[8

2]

>>> [

x+yfor x in vec1 for y in vec

-3]

, 5, -7, 8, 7, -5, 10, 9,

[6

1))]

vec1[i]\*vec2[i] for iin range(len(vec

>>> [

, 12,

[8

-54]



>>> [

str

(

round(355/113,

i

for

))

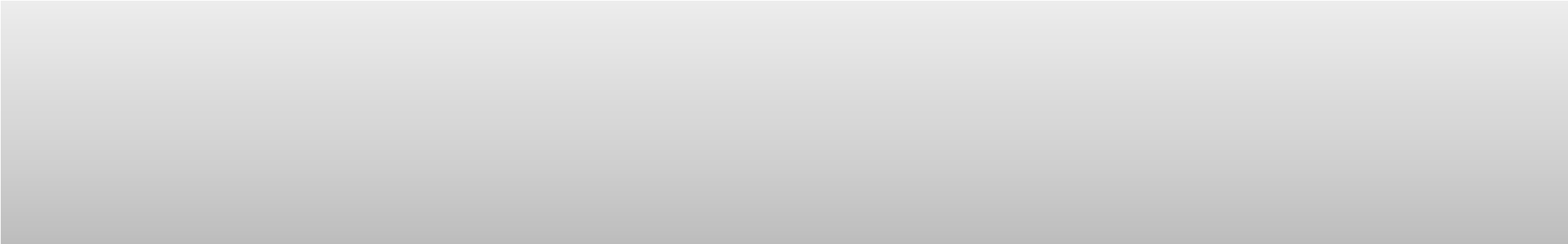
i

in range(1, 6)]

]

'3.1', '3.14', '3.142', '3.1416', '3.14159'

[



>>>

mat = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]

]

>>>

print([[row[i] for row in mat] for iin [0, 1,

2]])

[[1

, 4, 7], [2, 5, 8], [3, 6,

9]]

>>>

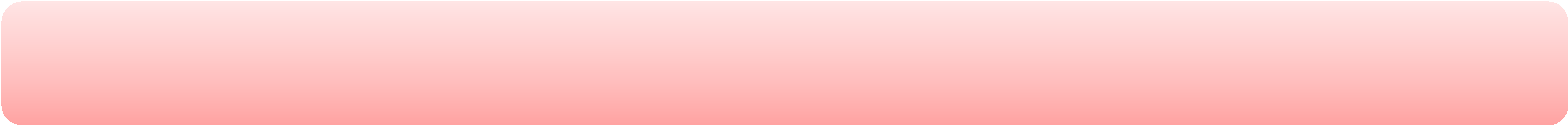
list(zip(\*mat

))

[(1

, 4, 7), (2, 5, 8), (3, 6,

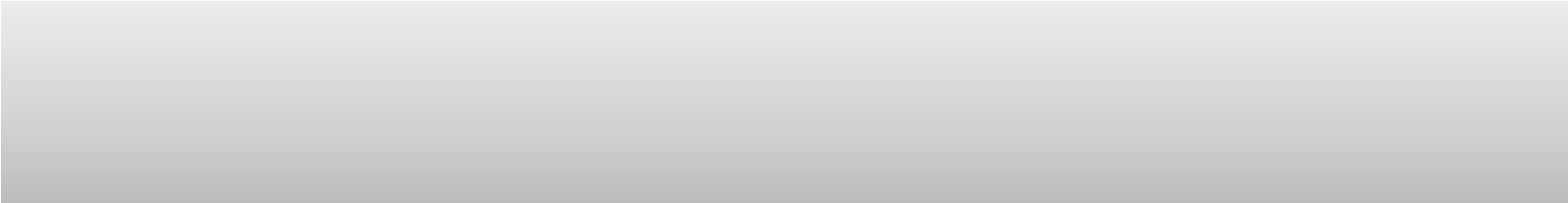
9)]



We will talk about \* during our discussion on fuctions

## Dictionary Comprehension

• A dictionary comprehension is like a list comprehension, but it constructs a dictionary instead of a list.



>>>

a\_dict= {'a': 1, 'b': 2, 'c':

3}

>>> {

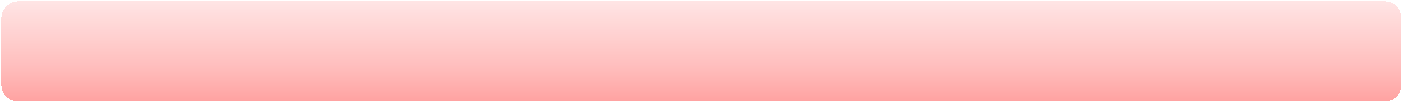
()}

value:keyfor key, value in a\_dict.items

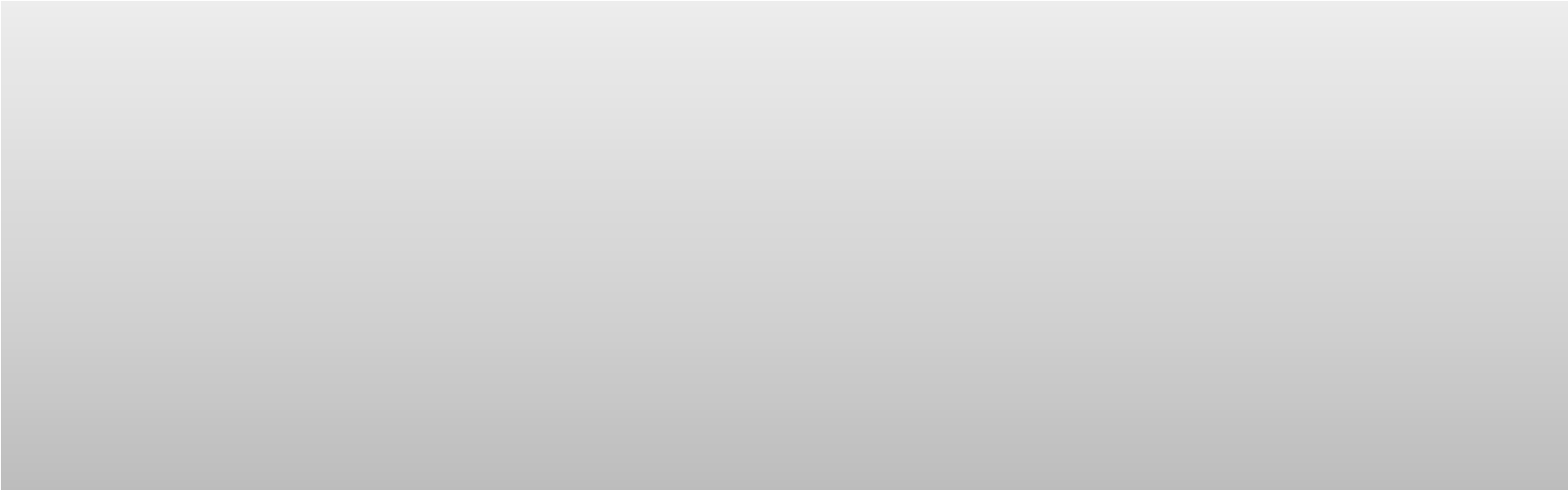
}

'a', 2: 'b', 3: 'c'

{1:



What are you trying to do here?



>>>

a\_dict= {'a': [1, 2, 3], 'b': 4, 'c':

5}

>>> {

value:keyfor key, value in a\_dict.items

()}

Traceback(most recent call last):

File "<pyshell#68>", line 1, in <module>

{

value:keyfor key, value in a\_dict.items

()}

File "<pyshell#68>", line 1, in <dictcomp>

{

value:keyfor key, value in a\_dict.items

()}

TypeError: unhashabletype: 'list'



Values of the dictionary are immutable, like string

s or tuples. If

you try this with a dictionary that contains lists,

it will fail .

## Set Comprehensions

* Sets have their comprehensions as well
* Syntax is similar to that of Dictionary comprehensions
* The only difference is that sets just have values instead of key:value pairs

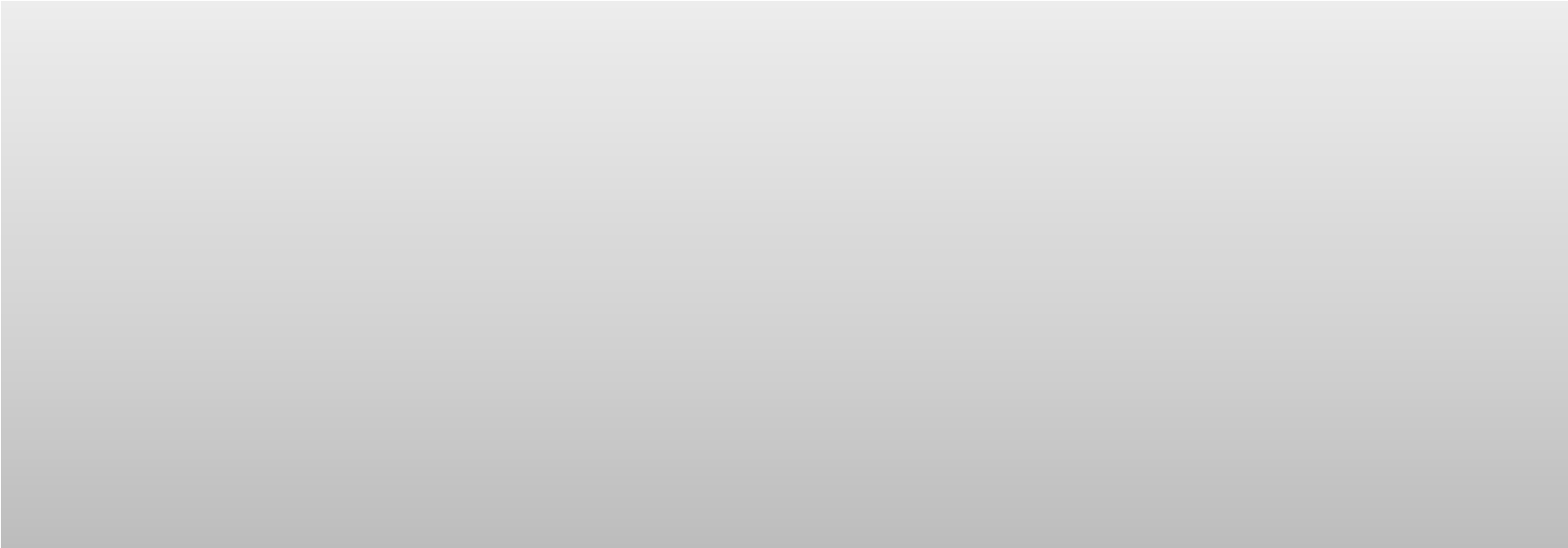
>>> a\_set = set(range(10))

>>> a\_set

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> {x \*\* 2 for x in a\_set}

{0, 1, 4, 81, 64, 9, 16, 49, 25, 36} >>> {x for x in a\_set if x % 2 == 0}



{0, 8, 2, 4, 6}

>>> {2\*\*x for x in range(10)}

{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}

## Use of Set Comprehension

Calculation of the prime numbers between 1 and 100 using the sieve of Eratosthenes:

>>> noprimes = [j for i in range(2, 8) for j in range(i\*2, 100, i)]

>>> primes = [x for x in range(2, 100) if x not in noprimes]

>>> print (primes)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,

79, 83, 89, 97]

Generalize the above problem so that we can use it with any arbitrary number n

>>> from math import sqrt >>> n = 100

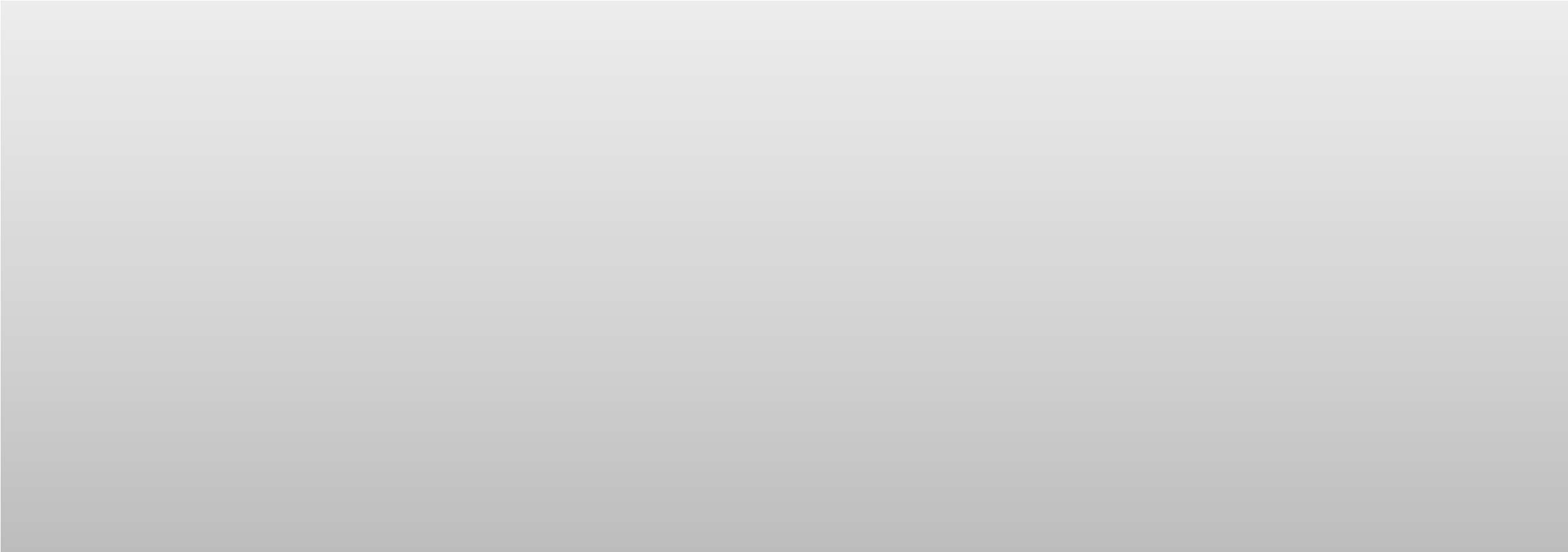
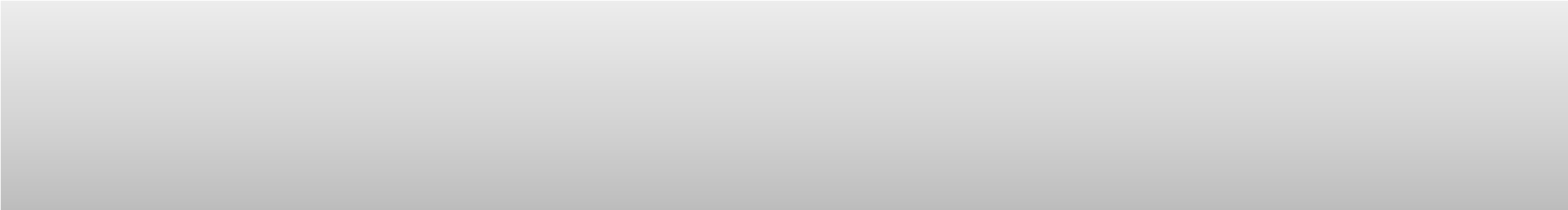
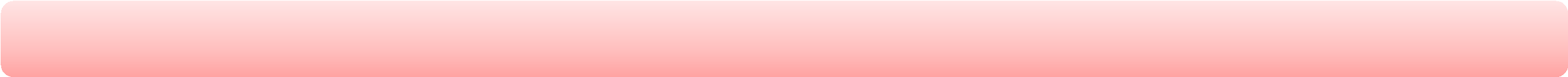
>>> sqrt\_n = int(sqrt(n))

>>> no\_primes = [j for i in range(2,sqrt\_n) for j in range(i\*2, n, i)]

>>> print(no\_primes)

[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,

44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39,



42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84,

88, 92, 96, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90,

95, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96, 14, 21, 28, 35,

42, 49, 56, 63, 70, 77, 84, 91, 98, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96,

18, 27, 36, 45, 54, 63, 72, 81, 90, 99]

What’s the problem? Double entries?

## Use of Set Comprehension

>>> from math import sqrt >>> n = 100

>>> sqrt\_n = int(sqrt(n))

>>> no\_primes = {j for i in range(2,sqrt\_n) for j in range(i\*2, n, i)}

>>> no\_primes

{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58,

60, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78, 80, 81, 82, 84, 85,

86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 98, 99}

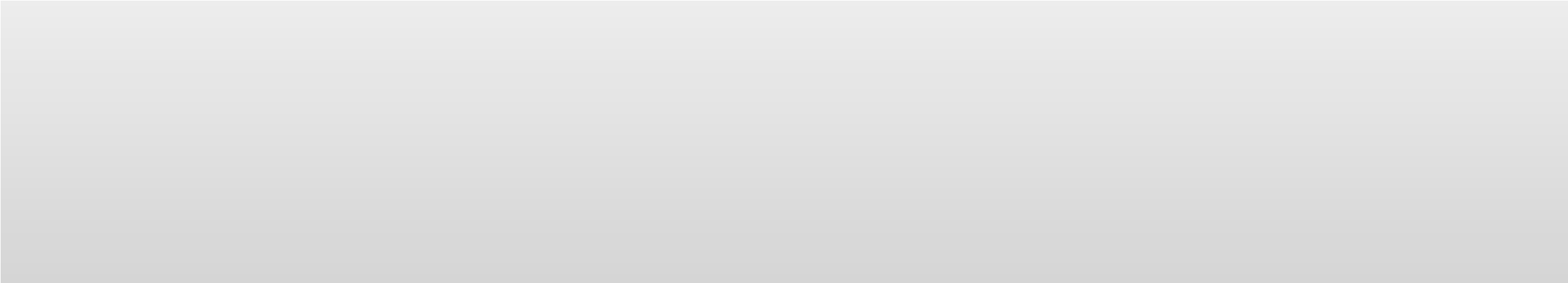
>>> primes = {i for i in range(n) if i not in no\_primes}

>>> print(primes)

{0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,

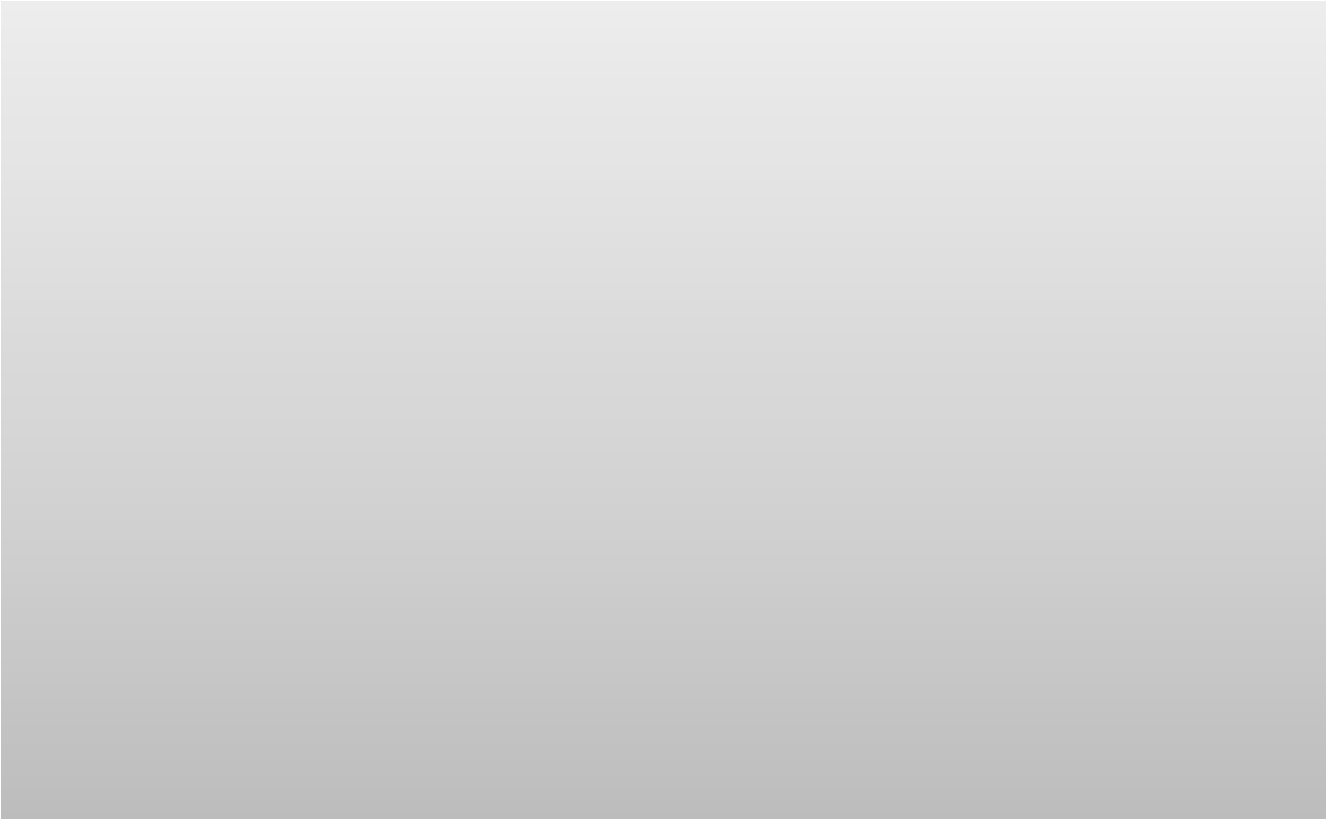
71, 73, 79, 83, 89, 97}

Solution: Set comprehension is the right functionality to solve our problem from the previous subsection. We are able to create the set of non primes without doublets



## Generators

* Generators are iterators, but you can only iterate over them once
* It’s because they do not store all the values in memory, they generate the values on the fly



>>>

generator = (x\*x for x in range

(3))

>>>

for iin generator

:

print(i)

0

1

4

>>>

for k in generator

:

print(k\*\*2)

>>>



Nothing printed. Why?

# enumerate()

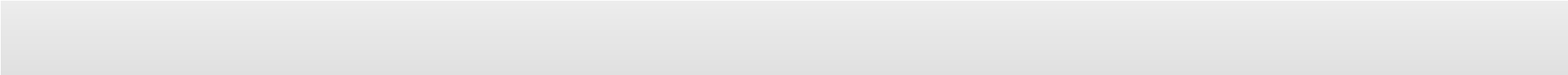
• Creates and enumerated object of an iterable

**enumerate(iterable, start=0)**

>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']

>>> list(enumerate(seasons))

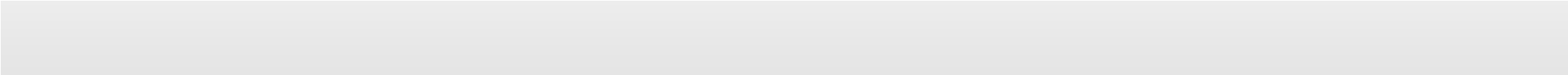
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')] >>> list(enumerate(seasons, start=1))



[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

# eval()

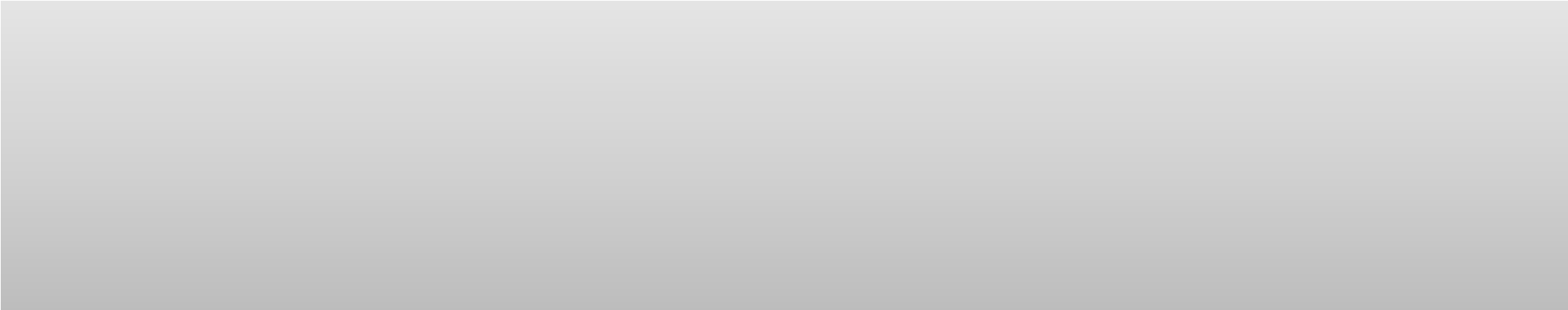
• Evaluates a given expression **eval(expression)**



>>>

x =

5



>>>

eval

(

x\*\*2 + x +

1)

Traceback(most recent call last):

File "<pyshell#78>", line 1, in <module>

eval(x\*\*2 + x + 1)

TypeError: eval() arg1 must be a string, bytes or code object

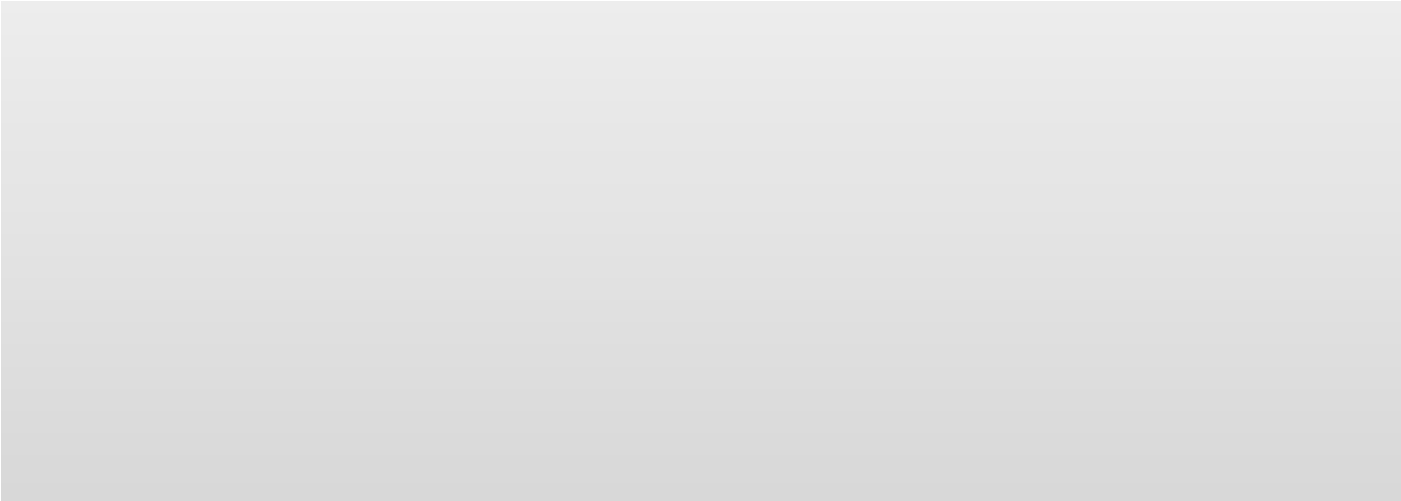
>>>

eval('x\*\*2 + x + 1'

)

31

## Shallow and Deep Copy



>>>

C1 = ['red', 'green', 'yellow'

]

>>>

C2 = ['cyan', 'purple', 'magenta'

]

C2 = C

1

>>>

>>>

C

1

[

]

'red', 'green', 'yellow'

2

C

>>>

]

'red', 'green', 'yellow'

[

C1[2] = 'blue'

>>>



Consider the following session



1

>>>

C

]

[

'red', 'green', 'blue'

>>>

C

2

[

'red', 'green', 'blue'

]

>>>

C2[0] = 'rouge'

>>>

C

1

[

'rouge', 'green', 'blue'

]

>>>

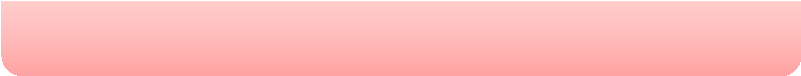
C

2

[

'rouge', 'green', 'blue'

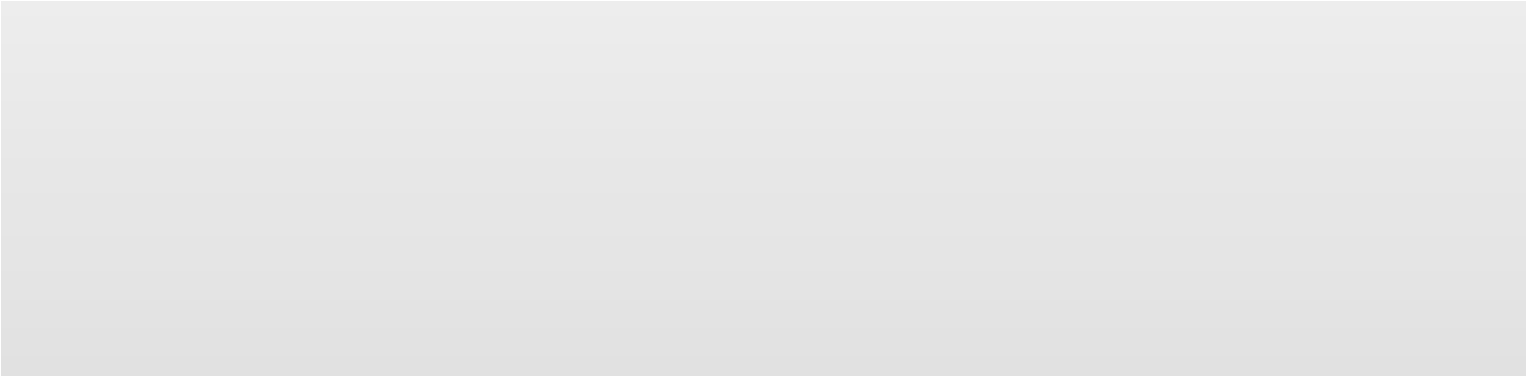
]



Automatically changed

## Shallow and Deep Copy

* The effect in the previous example is because of the shallow copy
* The effect can be mitigated using slice operator as shown, but has limitations



>>>

C1 = ['red', 'green', 'yellow'

]

]

>>>

C2 = ['cyan', 'purple', 'magenta'

>>>

C2 = C

1[:]

>>>

C

2

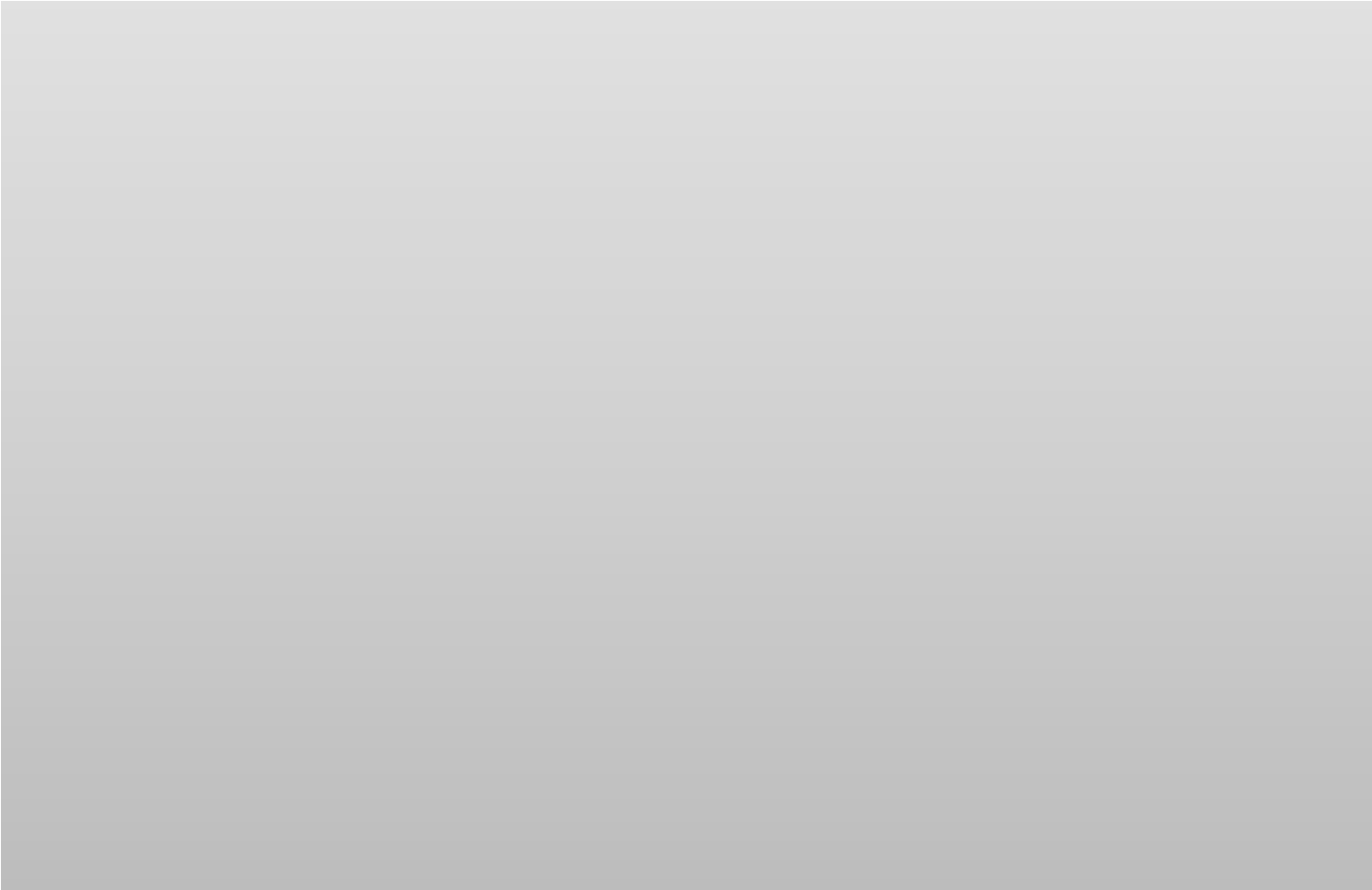
[

'red', 'green', 'yellow'

]

>>>

C2[1] = 'rouge'



2

>>>

C

[

'red', 'rouge', 'yellow'

]

1

>>>

C

[

'red', 'green', 'yellow'

]

>>>

C1 = ['red', 'green', ['blue', 'yellow'

]]

]

>>>

C2 = ['cyan', 'purple', 'magenta'

1[:]

C2 = C

>>>

2

C

>>>

]]

'red', 'green', ['blue', 'yellow'

[

C2[2][0] = 'bleu'

>>>

>>>

C2[2][1] = 'jaune'

>>>

C

2

]]

[

'red', 'green', ['bleu', 'jaune'

>>>

C

1

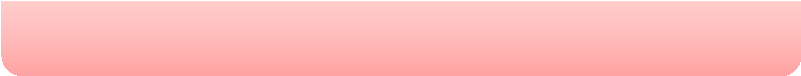
]]

[

'red', 'green', ['bleu', 'jaune'



Automatically changed



No Change

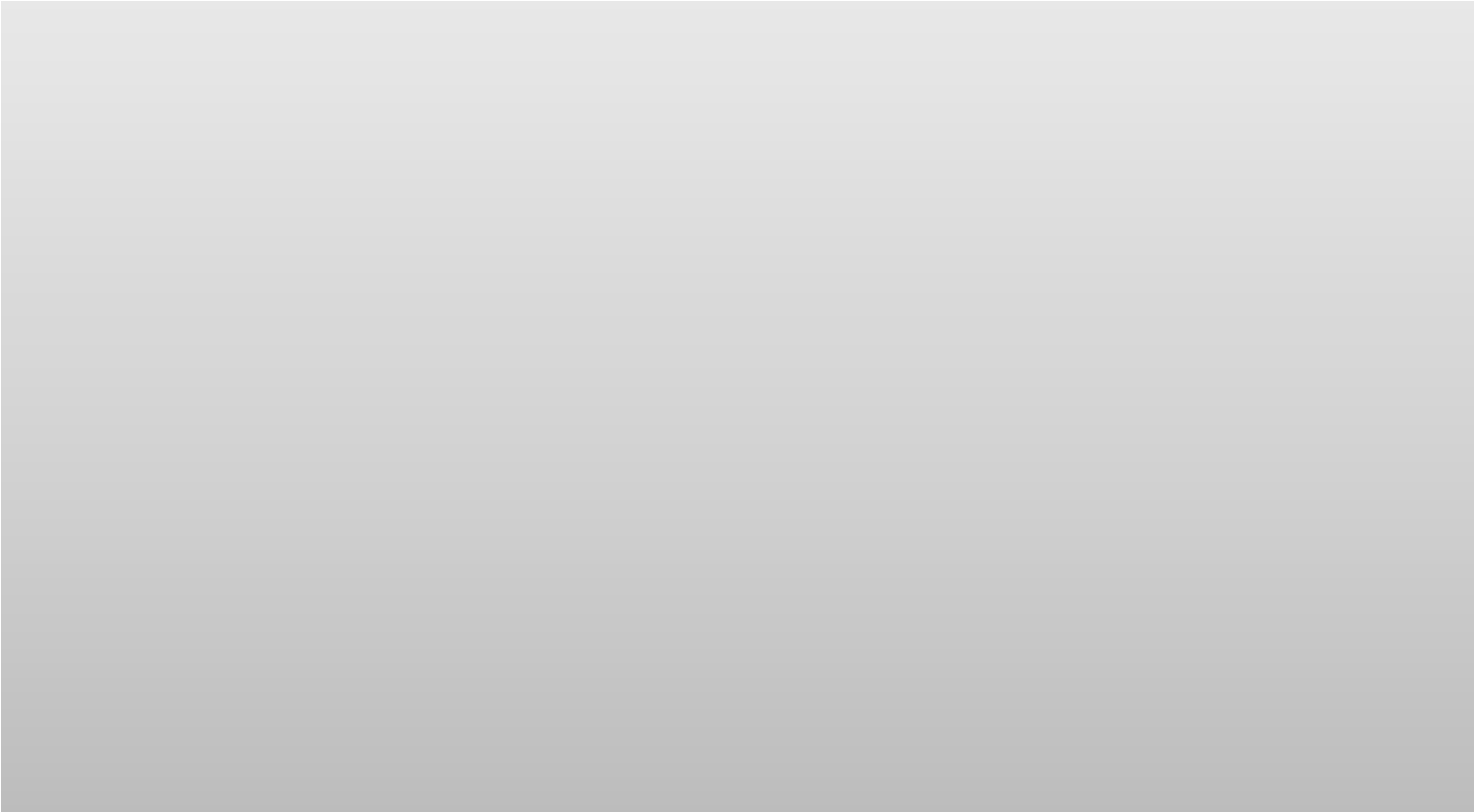
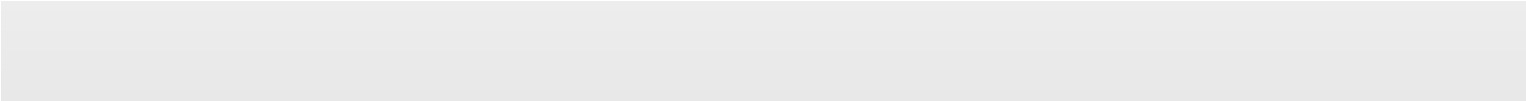
## Deep Copy using **deepcopy()**

• A solution to the described problems is to use the module "copy". This module provides the method "copy", which allows a complete copy of a arbitrary list, i.e. shallow and other lists.

>>> from copy import deepcopy

>>> C1 = ['red', 'green', ['blue', 'yellow']]

>>> C2 = ['cyan', 'purple', 'magenta'] >>> C1



['red', 'green', ['blue', 'yellow']]

>>> C2

['cyan', 'purple', 'magenta']

>>> C2 = deepcopy(C1)

>>> C2

['red', 'green', ['blue', 'yellow']]

>>> C2[2][1] = 'jaune'

>>> C1

['red', 'green', ['blue', 'yellow']]

>>> C2

['red', 'green', ['blue', 'jaune']]

## Copying: Summary

* Assignment statements in Python do not copy objects, they create bindings between a target and an object.
* For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other.
* **copy** module provides generic shallow and deep copy operations
  + **copy.copy(*x*)** – returns a shallow copy of x
  + **copy.deepcopy(*x*)** – returns a deep copy of x

# itertools

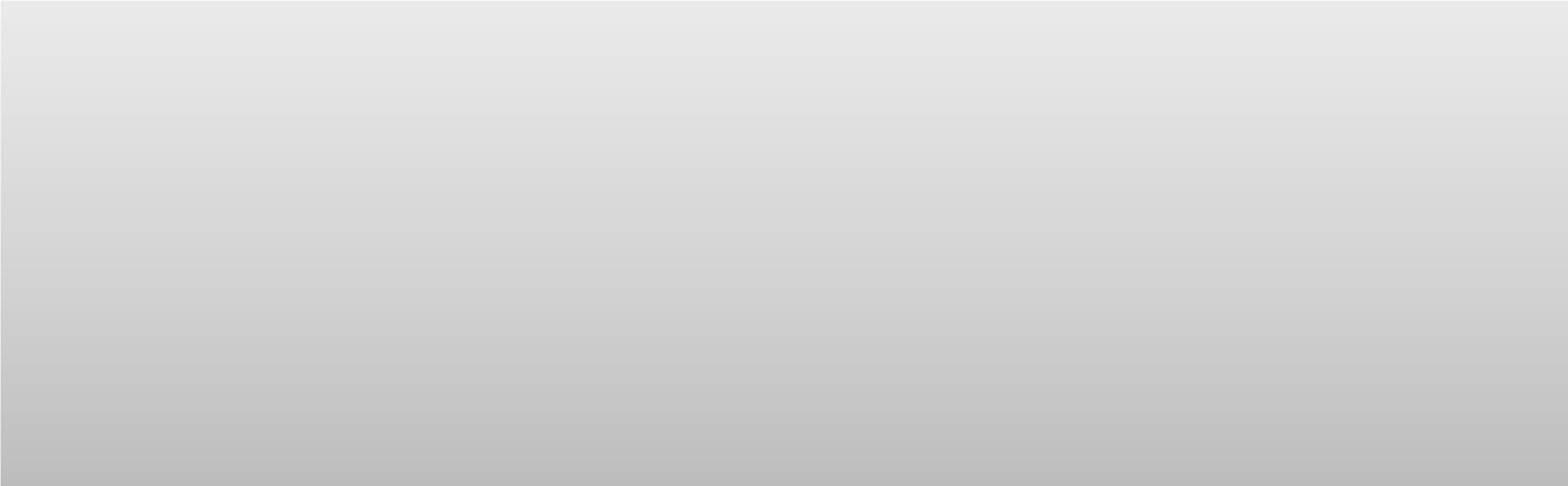
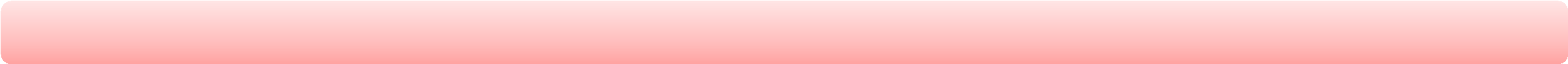
* **itertools** is a module that contains several functions for creating iterators for efficient looping
* Few examples are given in below interactive session:

>>> import itertools

>>> list(itertools.permutations('ATGC', 4))

[('A', 'T', 'G', 'C'), ('A', 'T', 'C', 'G'), ('A', 'G', 'T', 'C'), ('A', 'G', 'C',

'T'), ('A', 'C', 'T', 'G'), ('A', 'C', 'G', 'T'), ('T', 'A', 'G', 'C'), ('T', 'A', 'C', 'G'), ('T', 'G', 'A', 'C'), ('T', 'G', 'C', 'A'), ('T', 'C', 'A', 'G'), ('T',



'C', 'G', 'A'), ('G', 'A', 'T', 'C'), ('G', 'A', 'C', 'T'), ('G', 'T', 'A', 'C'),

('G', 'T', 'C', 'A'), ('G', 'C', 'A', 'T'), ('G', 'C', 'T', 'A'), ('C', 'A', 'T',

'G'), ('C', 'A', 'G', 'T'), ('C', 'T', 'A', 'G'), ('C', 'T', 'G', 'A'), ('C', 'G', 'A', 'T'), ('C', 'G', 'T', 'A')]

>>> list(itertools.combinations('ATGC', 4)) [('A', 'T', 'G', 'C')]

>>> list(itertools.combinations('ATGC', 2))

[('A', 'T'), ('A', 'G'), ('A', 'C'), ('T', 'G'), ('T', 'C'), ('G', 'C')]

>>>

Refer: https://docs.python.org/2/library/itertools.html

# itertools

>>> import itertools

>>> itertools.product('GACT', repeat = 2)

<itertools.product object at 0x01382AD0>

>>> list(itertools.product('GACT', repeat = 2))

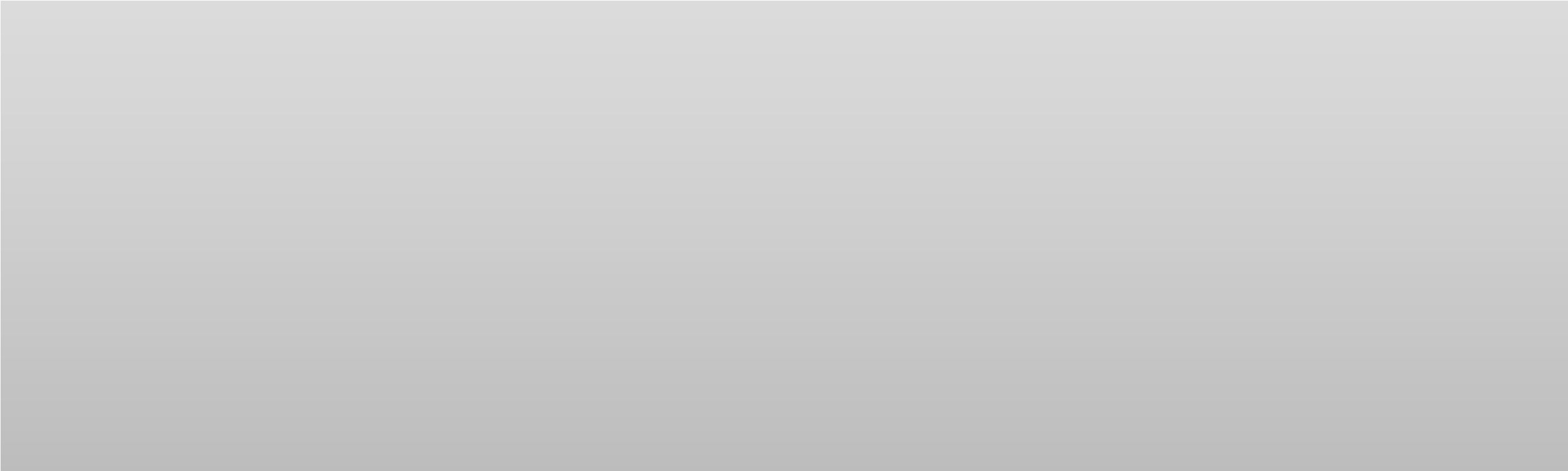
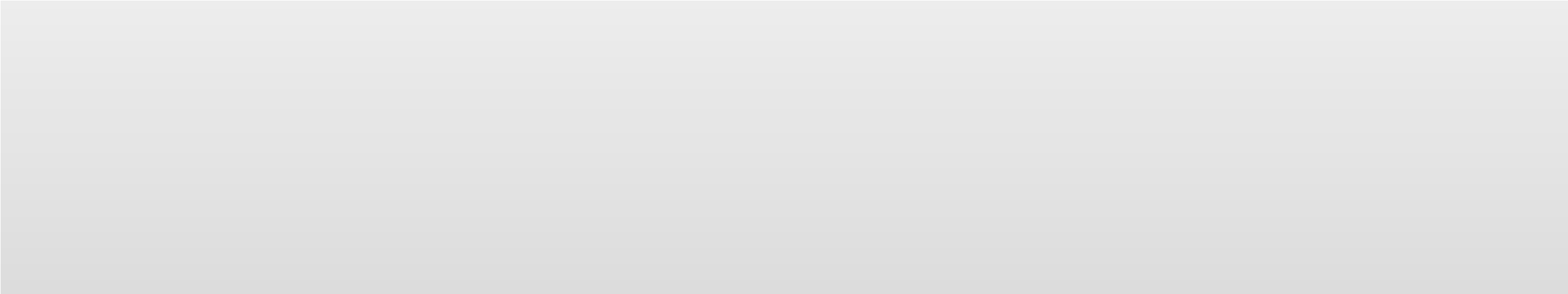
[('G', 'G'), ('G', 'A'), ('G', 'C'), ('G', 'T'), ('A', 'G'), ('A', 'A'), ('A', 'C'),

('A', 'T'), ('C', 'G'), ('C', 'A'), ('C', 'C'), ('C', 'T'), ('T', 'G'), ('T', 'A'),

('T', 'C'), ('T', 'T')]

>>> list(itertools.product('GACT', repeat = 3))

[('G', 'G', 'G'), ('G', 'G', 'A'), ('G', 'G', 'C'), ('G', 'G', 'T'), ('G', 'A', 'G'),



('G', 'A', 'A'), ('G', 'A', 'C'), ('G', 'A', 'T'), ('G', 'C', 'G'), ('G', 'C', 'A'),

('G', 'C', 'C'), ('G', 'C', 'T'), ('G', 'T', 'G'), ('G', 'T', 'A'), ('G', 'T', 'C'),

('G', 'T', 'T'), ('A', 'G', 'G'), ('A', 'G', 'A'), ('A', 'G', 'C'), ('A', 'G', 'T'),

('A', 'A', 'G'), ('A', 'A', 'A'), ('A', 'A', 'C'), ('A', 'A', 'T'), ('A', 'C', 'G'),

('A', 'C', 'A'), ('A', 'C', 'C'), ('A', 'C', 'T'), ('A', 'T', 'G'), ('A', 'T', 'A'),

('A', 'T', 'C'), ('A', 'T', 'T'), ('C', 'G', 'G'), ('C', 'G', 'A'), ('C', 'G', 'C'),

('C', 'G', 'T'), ('C', 'A', 'G'), ('C', 'A', 'A'), ('C', 'A', 'C'), ('C', 'A', 'T'),

('C', 'C', 'G'), ('C', 'C', 'A'), ('C', 'C', 'C'), ('C', 'C', 'T'), ('C', 'T', 'G'),

('C', 'T', 'A'), ('C', 'T', 'C'), ('C', 'T', 'T'), ('T', 'G', 'G'), ('T', 'G', 'A'),

('T', 'G', 'C'), ('T', 'G', 'T'), ('T', 'A', 'G'), ('T', 'A', 'A'), ('T', 'A', 'C'),

('T', 'A', 'T'), ('T', 'C', 'G'), ('T', 'C', 'A'), ('T', 'C', 'C'), ('T', 'C', 'T'),

('T', 'T', 'G'), ('T', 'T', 'A'), ('T', 'T', 'C'), ('T', 'T', 'T')]

## Challenge #1

* Given an array of random numbers, print the numbers and the number of occurrences of each number in a neatly formatted table
* Improvise it for: given a paragraph in English, write a python script to print the histogram of words

## Mini-Project

• Given a date and time, write a python script to display the elements of the Vedic calendar and the planetary positions

Do your research on the astrological calculations

