National University of Computer and Emerging Sciences



Lab Manual 02

Artificial Intelligence Lab

Department of Computer Science

FAST-NU, Lahore, Pakistan

# Python Lists

Lists in Python represent ordered sequences of values. Lists are "mutable". You can access individual list elements with square brackets. Python uses *zero-based* indexing, so the first element has index 0.

Here are a few examples of how to create lists:

# List of integers

primes = [2, 3, 5, 7]

# We can put other types of things in lists

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

# We can even make a list of lists

hands = [

['J', 'Q', 'K'],

['2', '2', '2'],

['6', 'A', 'K'], # (Comma after the last element is optional)

]

# A list can contain a mix of different types of variables:

my\_favourite\_things = [32, 'AI Lab’, 100.25]

## Indexing & Slicing Examples

Consider our list of planets created above:

planets[0] # 'Mercury'

planets[1] # 'Venus'

planets[-1] # 'Neptune'

planets[-2] # 'Uranus'

# List Slicing

# first three planets

planets[0:3] # ['Mercury', 'Venus', 'Earth']

planets[:3] # ['Mercury', 'Venus', 'Earth']

# All the planets from index 3 onward

planets[3:] # ['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

# All the planets except the first and last

planets[1:-1] # ['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']

# The last 3 planets

planets[-3:] # ['Saturn', 'Uranus', 'Neptune']

## List Modification Examples

Working with the same planets list:

# Rename Mars

planets[3] = 'Malacandra'

# ['Mercury', 'Venus', 'Earth', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

# Rename multiple list indexes

planets[:3] = ['Mur', 'Vee', 'Ur']

['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

## List functions

Python has several useful functions for working with lists.

len(planets) # 8

# The planets sorted in alphabetical order

sorted(planets)

# ['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']

nums = [9,8,7,6,5,4,3,2,1,0]

sorted(nums)

primes = [2, 3, 5, 7]

sum(primes) # 17

max(primes) # 7

min(primes)

# Let’s add Pluto to the planets list

planets.append('Pluto')

# Pop removes and returns the last element of the list

planets.pop() # ‘Pluto’

# Remove an item from a list given its index instead of its value

a = [-1, 1, 66.25, 333, 333, 1234.5]

del a[0] # [1, 66.25, 333, 333, 1234.5]

# Remove slices from the list

del a[2:4] # [1, 66.25, 1234.5]

planets.index('Earth') # 2

# Finally to find all the methods associated with Python list object

help(planets)

# 

# 2. Python Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

1. The syntax for creating them uses parentheses instead of square brackets.
2. They cannot be modified (they are *immutable*).

**Tuples can be indexed and sliced just like lists.**

t = (1, 2, 3)

t = 1, 2, 3 # equivalent to above

t[0] = 100 # TypeError: 'tuple' object does not support item assignment

## Tuple Functions

There are only two tuple methods count() and index() that a tuple object can call.

thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)  
x = thistuple.count(5) # 2

thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

x = thistuple.index(8) # 3

# 

# Python Sets

* A set is a collection which is *unordered*, *unchangeable*, and *unindexed*.
* Unordered means that the items in a set do not have a defined order.Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
* Set items are unchangeable, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot modify its items, but you can remove items and add new items.
* Sets cannot have two items with the same value.
* In particular, list cannot be an element of a set (but tuple can), and another set cannot be an element of a set.

## Set Initialization Examples

You can initialize a set in the following ways:

# Initialize empty set

emptySet = set()

# Pass a list to set() to initialize it

dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])

dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])

# Direct initialization using curly braces

dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}

dataEngineer = {'Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'}

# Curly braces can only be used to initialize a set containing values

emptyDict= {} # type(emptyDict) is a dictionary

## Set Modification Examples

Let’s consider the following set for our add/remove examples:

# Initialize set with values

graphicDesigner = {'InDesign', 'Photoshop', 'Acrobat', 'Premiere', 'Bridge'}

# Add a new immutable element to the set

graphicDesigner.add('Illustrator')

# TypeError: unhasable type ‘list’

graphicDesigner.add(['Powerpoint', 'Blender'])

# List cannot be added to a set

# Remove an element from the set

graphicDesigner.remove('Illustrator')

# Another way to remove an element. What is the difference?

graphicDesigner.discard('Premiere')

**# The discard() method removes the specified item from the set**. This method is different from the remove() method, because the remove() method will raise an error if the specified item does not exist, and the discard() method will not.

# Remove and return an arbitrary value from a set, pop removes the first element from the set

graphicDesigner.pop()

# Remove all values from the set, set becomes empty

graphicDesigner.clear()

# 

# 

# Python Dictionaries

Dictionaries and lists share the following characteristics:

* Both are mutable **(values are mutable only, not keys)**
* Both are dynamic. They can grow and shrink as needed.
* Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

* List elements are accessed by their position in the list, via indexing.
* Dictionary elements are accessed via keys not by numerical index.

Duplicate keys are not allowed. A dictionary key must be of a type that is immutable.

Here are a few examples to create dictionaries:

MLB\_team = {

'Colorado' : 'Rockies',

'Boston' : 'Red Sox',

'Minnesota': 'Twins',

'Milwaukee': 'Brewers',

'Seattle' : 'Mariners'

}

# Can also be defined as:

MLB\_team = dict([

('Colorado', 'Rockies'),

('Boston', 'Red Sox'),

('Minnesota', 'Twins'),

('Milwaukee', 'Brewers'),

('Seattle', 'Mariners')

])

# Another way

tel = dict(sape=4139, guido=4127, jack=4098)

# Building a dictionary incrementally – if you don’t know all the key-value pairs in advance

person = {}

person['fname'] = 'Joe'

person['lname'] = 'Fonebone'

person['age'] = 51

person['spouse'] = 'Edna'

person['children'] = ['Ralph', 'Betty', 'Joey']

person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}

# {'fname': 'Joe', 'lname': 'Fonebone', 'age': 51, 'spouse': 'Edna',

'children': ['Ralph', 'Betty', 'Joey'], 'pets': {'dog': 'Fido', 'cat': 'Sox'}}

## Dictionary Modification Examples

A few examples to access the dictionary elements, add new key value pairs, or update previous value:

# Retrieve a value

MLB\_team['Minnesota'] # 'Twins'

# Add a new entry

MLB\_team['Kansas City'] = 'Royals'

# Update an entry

MLB\_team['Seattle'] = 'Seahawks'

## Dictionary Functions

The following is an overview of methods that apply to dictionaries:

# Let’s use this dict to demonstrate dictionary functions

d = {'a': 10, 'b': 20, 'c': 30}

# Clears a dictionary.

d.clear() # {}

# Returns the value for a key if it exists in the dictionary.

print(d.get('b')) # 20

# Removes a key from a dictionary, if it is present, and returns its value.

d.pop('b') # 20

# Returns a list of key-value pairs in a dictionary.

list(d.items()) # [('a', 10), ('b', 20), ('c', 30)]

list(d.items())[1][0] # 'b'

list(d.items())[1][1] # 20

# Returns a list of keys in a dictionary.

list(d.keys()) # ['a', 'b', 'c']

# Returns a list of values in a dictionary.

list(d.values()) # [10, 20, 30]

# Removes the last key-value pair from a dictionary.

d.popitem() # ('c', 30)

# Merges a dictionary with another dictionary or with an iterable of key-value pairs.

d2 = {'b': 200, 'd': 400}

d.update(d2) # {'a': 10, 'b': 200, 'c': 30, 'd': 400}

For more details, visit [iterate dictionary](https://realpython.com/iterate-through-dictionary-python/) & [dictionary comprehensions](https://www.datacamp.com/community/tutorials/python-dictionary-comprehension).

**Classes in python:**

In Python, classes are a fundamental concept of object-oriented programming (OOP). They provide a way to structure and organize code by grouping related data and functionality together. Here's a basic overview of how classes work in Python:

**Class Definition:**

You define a class using the class keyword, followed by the class name. Inside the class, you can define attributes and methods.

#code

Example card

class MyClass:  
 attribute1 = "value1"  
 attribute2 = "value2"  
  
obj1 = MyClass()  
obj2 = MyClass()  
  
print(obj1.attribute1) *# Output: value1*obj2.attribute1 = "new\_value"  
print(obj1.attribute1) *# Output: value1 (unchanged because obj2 created an instance attribute)*print(obj2.attribute1) *# Output: new\_value (instance attribute specific to obj2)*

#output

value1

value1

new\_value

**Define and Initialize variable in constructor.**

class MyClassWithConstructor:  
 def \_\_init\_\_(self, instance\_param):  
 self.instance\_attribute = instance\_param  
  
obj1 = MyClassWithConstructor("value1")  
obj2 = MyClassWithConstructor("value2")  
  
print(obj1.instance\_attribute) *# Output: value1*print(obj2.instance\_attribute)

#output

value1

value1

new\_value

**Differences between defining and initializing variables within a class versus inside the constructor in Python.**

This creates class variables, shared by all instances of the class. They exist independently of objects.

You can change any class variable value using: ClassName.class\_variable\_name = “AI”

**Inside the constructor (\_\_init\_\_):**

This defines instance variables, unique to each instance. They become part of the instance's state.

Usage: object\_name.instance\_variable\_name.

**How to make variable private in class:**

class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.\_balance = balance #private variable  
  
 def get\_balance(self):  
 return self.\_balance  
  
 def deposit(self, amount):  
 self.\_balance += amount  
  
 def withdraw(self, amount):  
 if self.\_balance >= amount:  
 self.\_balance -= amount  
 else:  
 print("Insufficient funds")

obj=BankAccount(2000)  
print(obj.get\_balance())  
obj.deposit(500)  
print(obj.get\_balance())  
obj.withdraw(200)  
print(obj.get\_balance())

#output

2000

2500

2300

# 