**USING LIST AND DICTIONARIES – DATABASE**

**LISTS – EXAMPLE\_1 (using python interpreter)**

Lists, for example, can collect attributes about people in a positionally ordered way.

Create two records:

>>> bob = ['Bob Smith', 42, 30000, 'software']

>>> sue = ['Sue Jones', 45, 40000, 'hardware']

Each record is a list of four properties: name, age, pay, and job fields.

**OPERATIONS ON THESE RECORDS**

**TO ACCESS INDIVIDUAL FIELDS** – Index by position

>>> bob[0], sue[2] # fetch name, pay

('Bob Smith', 40000)

**TO EXTRACT LAST NAME: -** split and indexing

>>> bob[0].split()[-1] # what's bob's last name?

'Smith'

**Details**

>>> bob[0].split()

['Bob', 'Smith']

>>> bob[0].split()[-1]

'Smith'

**MULTIPLICATION**

>>> sue[2] \*= 1.25 # give sue a 25% raise

**Otherway?**

>>> sue[2] = sue[2] \* 1.25

**DISPLAY**

>>> sue

['Sue Jones', 45, 50000.0, 'hardware']

**A DATABASE LIST**

>>> people = [bob, sue] # reference in list of lists

**PRINT PEOPLE DATABASE**

>>> for person in people:

print(person)

**Details**

>>> for person in people:

... print(people)

File "<stdin>", line 2

print(people)

^

IndentationError: expected an indented block

>>> for person in people:

... print(person)

...

['Bob Smith', 42, 30000, 'software']

['Sue jones', 45, 50000.0, 'hardware']

**ACCESSING INDIVIDUAL FIELDS OF DATABASE RECORDS**

>>> people[1][0]

'Sue Jones'

**Details**

>>> people[1]

['Sue jones', 45, 50000.0, 'hardware']

>>> people[1][0]

'Sue jones'

>>> people[1][1]

45

>>> people[1][2]

50000.0

>>> people[1][3]

'hardware'

>>> for person in people:

print(person[0].split()[-1]) # print last names

person[2] \*= 1.20 # give each some raise

Smith

Jones

>>> for person in people: print(person[2]) # check new pay

36000.0

60000.0

**TO ADD RECORDS INTO DATABSE**

>>> people.append(['Tom', 50, 0, None])

>>> len(people)

3

>>> people[-1][0]

'Tom'

The list-based record representations in the prior example required to search for field names. This has some performance impact. There are more efficient and convenient ways to associate property names and values. The built-in dictionary object is a natural:

**DICTIONARIES – EXAMPLE\_2 (using python interpreter)**

>>> bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}

>>> sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}

Now,

* Bob and Sue are objects that map field names to values automatically, and
* they make our code more understandable and meaningful.
* We don’t have to remember what a numeric offset means, and
* we let Python search for the value associated with a field’s name with its efficient dictionary indexing.

>>> bob['name'], sue['pay’] # not bob[0], sue[2]

('Bob Smith', 40000)

>>> bob['name'].split()[-1]

'Smith'

>>> sue['pay'] \*= 1.10

>>> sue['pay']

44000.0

>>> people = [bob, sue]

>>> for person in people:

... print(person['name'], person['pay'])

...

('bob smith', 30000)

('sue Jones', 40000)