Using List

Start-up pointers

Lists can collect attributes about people in a positionally ordered way.represent two people, Bob and Sue(name, age, pay, and job fields)

In [1]:
```
bob = ['Bob Smith', 42, 30000, 'software']
sue = ['Sue Jones', 45, 40000, 'hardware']
```

In [2]:
```
bob[0],sue[2] # fetch bob's name, sue's pay
```

Out[2]: ('Bob Smith', 40000)

What's bob's last name?

In [3]:
```
bob[0].split()[-1] # bob's last name
```

Out[3]: 'Smith'

Give sue a 25% raise!

In [4]:
```
sue[2] *= 1.25 # raise 25%
sue
```

Out[4]: ['Sue Jones', 45, 50000.0, 'hardware']

A Database list

To collect Bob and Sue into a unit, we might simply stuff them into another list.

In [5]:
```
people = [bob,sue] # reference in list of lists
for person in people: # loop
    print(person)
```

```
['Bob Smith', 42, 30000, 'software']
['Sue Jones', 45, 50000.0, 'hardware']
```

Now the people list represents our database. We can fetch specific records by their relative positions and process them one at a time, in loops:

In [6]:
```
people[1][0]
```

Out[6]:
```
'Sue Jones'
```

In [7]:
```
for person in people:
    print(person[0].split()[-1]) # print last name
    person[2] *= 1.20            # give each a 20% raise
```

```
Smith
Jones
```

In [8]:
```
for person in people: print(person[2]) # check new pay
```

```
36000.0
60000.0
```

In [9]:
```
pays = [person[2] for person in people] # collect all pay
pays
```

Out[9]:
```
[36000.0, 60000.0]
```

In [10]:
```
pays = map((lambda x: x[2]), people)  # ditto (map is a generator in 3.X)
list(pays)
```

Out[10]:
```
[36000.0, 60000.0]
```

In [11]:
```
sum(person[2] for person in people)   # generator expression, sum built-in
```

Out[11]:
```
96000.0
```

To add a record to the database, the usual list operations, such as append and extend, will suffice:

In [12]:
```
people.append(['Tom', 50, 0, None])
len(people)
```

Out[12]:
```
3
```

In [13]:
```
people[-1][0]
```

Out[13]:
```
'Tom'
```

Lists work for our people database, and they might be sufficient for some programs, but they suffer from a few major flaws. For one thing, Bob and Sue, at this point, are just fleeting objects in memory that will disappear once we exit Python. For another, every time we want to extract a last name or give a raise, we'll have to repeat the kinds of code we just typed; that could become a problem if we ever change the way those operations work ——we may have to update many places in our code. We'll address these issues in a few moments.

Field labels

Perhaps more fundamentally, accessing fields by position in a list requires us to memorize what each position means: if you see a bit of code indexing a record on magic position 2, how can you tell it is extracting a pay? In terms of understanding the code, it might be better to associate a field name with a field value.

In [14]:
```
NAME, AGE, PAY = range(3) # 0, 1 and 2
bob = ['Bob Smith', 42, 10000]
bob[NAME]
PAY,bob[PAY]
```

Out[14]: (2, 10000)

This addresses readability: the three uppercase variables essentially become field names. This makes our code dependent on the field position assignments, though we have to remember to update the range assignments whenever we change record structure. Because they are not directly associated, the names and records may become out of sync over time and require a maintenance step.

We might also try this by using lists of tuples, where the tuples record both a field nameand a value; better yet, a list of lists would allow for updates (tuples are immutable).Here's what that idea translates to, with slightly simpler records:

In [15]:
```
bob = [['name', 'Bob Smith'], ['age', 42], ['pay', 10000]]
sue = [['name', 'Sue Jones'], ['age', 45], ['pay', 20000]]
people = [bob, sue]

for person in people:
    print(person[0][1], person[2][1]) # print name, pay
```

```
Bob Smith 10000
Sue Jones 20000
```

In [16]:
```
names = [person[0][1] for person in people] # collect name
names
```

Out[16]: ['Bob Smith', 'Sue Jones']

In [17]:
```
for person in people:
    print(person[0][1].split()[-1]) # get last name
    person[2][1] *= 1.10            # give a 10% raise

for person in people: print(person[2])
```

```
Smith
Jones
['pay', 11000.0]
['pay', 22000.0]
```

All we've really done here is add an extra level of positional indexing. To do better, we might inspect field names in loops to find the one we want (the loop uses tuple assignment here to unpack the name/value pairs):

In [18]:
```
for person in people:
    for (name, value) in person:
        if name == 'name':
            print(value) # find a specific field
```

```
Bob Smith
Sue Jones
```

Better yet, we can code a fetcher function to do the job for us:

In [19]:
```
def field(record, label):
    for (fname, fvalue) in record:
        if fname == label: # find a field by namme
            return fvalue
```

In [20]:
```
field(bob, 'name')
```

Out[20]: 'Bob Smith'

In [21]:
```
field(sue, 'pay')
```

Out[21]: 22000.0

In [22]:
```
for rec in people:
    print(field(rec, 'age')) # print all ages
```

```
42
45
```

Using Dictionaries

The list-based record representations in the prior section work, though not without some cost in terms of performance required to search for field names (assuming you need to care about milliseconds and such). But if you already know some Python, you also know that there are more efficient and convenient ways to associate property names and values. The built-in dictionary object is a natural:

In [23]:
```
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**:

In [24]:
```
bob['name'], sue['pay'] # not bob[0], sue[2]
```

Out[24]: ('Bob Smith', 40000)

In [25]:
```
bob['name'].split()[-1]
```

Out[25]: 'Smith'

In [26]:
```
sue['pay'] *= 1.10 # raise a 10%
sue['pay']
```

Out[26]: 44000.0

Because fields are accessed mnemonically now, they are more meaningful to those who read your code (including you).

Other ways to make dictionaries

Dictionaries turn out to be so useful in Python programming that there are even more convenient ways to code them than the traditional literal syntax shown earlier—e.g.,with keyword arguments and the type constructor, as long as the keys are all strings:

In [27]:
```
bob = dict(name='Bob Smith', age=42, pay=30000, jpb='dev')
sue = dict(name='Sue Jones', age=45, pay=40000, jpb='hdw')
```

In [28]:
```
bob
```

Out[28]: {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'jpb': 'dev'}

In [29]:
```
sue
```

Out[29]: {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'jpb': 'hdw'}

by filling out a dictionary one field at a time (recall that dictionary keys are pseudo-randomly ordered):

In [30]:
```
sue = {}
```

In [31]:
```
sue['name'] = 'Sue Jones'
sue['age'] = 45
sue['pay'] = 40000
sue['job'] = 'hdw'
```

In [32]:
```
sue
```

Out[32]: {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}

and by zipping together name/value lists:

In [33]:
```
names = ['name', 'age', 'pay', 'job']
values = ['Sue Jones', 45, 40000, 'hdw']
list(zip(names, values))
```

Out[33]: [('name', 'Sue Jones'), ('age', 45), ('pay', 40000), ('job', 'hdw')]

In [34]:
```
sue = dict(zip(names, values))
sue
```

Out[34]: {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}

We can even make dictionaries from a sequence of key values and an optional starting value for all the keys (handy to **initialize** an empty dictionary):

In [35]:
```
fields = ('name', 'age', 'job', 'pay')
record = dict.fromkeys(fields, '?')

record
```

Out[35]: {'name': '?', 'age': '?', 'job': '?', 'pay': '?'}

Lists of dictionaries

Regardless of how we code them, we still need to collect our dictionary-based records into a database; a list does the trick again, as long as we don't require access by key at the top level:

In [36]:
```
bob
```

Out[36]: {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'jpb': 'dev'}

In [37]:
```
sue
```

Out[37]: {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}

In [38]:
```
people = [bob, sue]                                  # reference in a list
for person in people:
    print(person['name'], person['pay'], sep=',')    # all name, pay
```

```
Bob Smith,30000
Sue Jones,40000
```

In [39]:
```
for person in people:
    if person['name'] == 'Sue Jones':                # fetch sue's pay
        print(person['pay'])
```

```
40000
```

Iteration tools work just as well here, but we use keys rather than obscure positions (in database terms, the list comprehension and map in the following code project the database on the “name” field column):

```
In [40]: names = [person['name'] for person in people]    # collect names
         names
Out[40]: ['Bob Smith', 'Sue Jones']

In [41]: list(map((lambda x: x['name']), people))         # ditto, generate
Out[41]: ['Bob Smith', 'Sue Jones']

In [42]: sum(person['pay'] for person in people)          # sum all pay
Out[42]: 70000
```

Interestingly, tools such as list comprehensions and on-demand generator expressions can even approach the utility of SQL queries here, albeit operating on in-memory objects:

```
In [43]: [rec['name'] for rec in people if rec['age'] >= 45] # SQL-ish query
Out[43]: ['Sue Jones']

In [44]: [(rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people]
Out[44]: [42, 2025]

In [45]: G = (rec['name'] for rec in people if rec['age'] >= 45)
         G
Out[45]: <generator object <genexpr> at 0x0000021461B369E8>

In [46]: next(G)
Out[46]: 'Sue Jones'

In [47]: G = ((rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people)
         G
Out[47]: <generator object <genexpr> at 0x0000021461B36FC0>

In [48]: G.__next__()
Out[48]: 42
```

And because dictionaries are normal Python objects, these records can also be accessed and updated with normal Python syntax:

In [49]:
```
for person in people:
    print(person['name'].split()[-1]) # last name
    person['pay'] *= 1.10
```

```
Smith
Jones
```

In [50]:
```
for person in people: print(person['pay'])
```

```
33000.0
44000.0
```

Nested structures

we could avoid the last-name extraction code in the prior examples by further structuring our records. Because all of Python's compound datatypes can be nested inside each other and as deeply as we like, we can build up fairly complex information structures easily—simply type the object's syntax, and Python does all the work of building the components, linking memory structures, and later reclaiming their space.

The following, for instance, represents a more structured record by nesting a **dictionary**, **list**, and **tuple** inside another dictionary:

In [51]:
```
bob2 = {'name':{'first': 'Bob', 'last': 'Smith'},
        'age': 42,
        'job': ['software', 'writing'],
        'pay': (40000, 50000)}
```

In [52]:
```
bob2['name']                          # bob's full name
```

Out[52]:
```
{'first': 'Bob', 'last': 'Smith'}
```

In [53]:
```
bob2['name']['last']                  # bob's last name
```

Out[53]:
```
'Smith'
```

In [54]:
```
bob2['pay'][1]                        # bob's upper pay
```

Out[54]:
```
50000
```

The name field is another dictionary here, so instead of splitting up a string, we simply index to fetch the last name. Moreover, people can have many jobs, as well as minimum and maximum pay limits. In fact, Python becomes a sort of query language in such cases—we can fetch or change nested data with the usual object operations:

In [55]:
```
for job in bob2['job']: print(job)      # all of bob's jobs
```

```
software
writing
```

In [56]:
```
bob2['job'][-1]                         # bob's last job
```

Out[56]:
```
'writing'
```

In [57]:
```
bob2['job'].append('janitor')           # bob's gets a new job
bob2
```

Out[57]:
```
{'name': {'first': 'Bob', 'last': 'Smith'},
 'age': 42,
 'job': ['software', 'writing', 'janitor'],
 'pay': (40000, 50000)}
```

It's OK to grow the nested list with append , because it is really an independent object. Such nesting can come in handy for more sophisticated applications; to keep ours simple, we'll stick to the original flat record structure.

Dictionaries of dictionaries

One last twist on our people database: we can get a little more mileage out of dictionaries here by using one to represent the database itself. That is, we can use a **dictionary of dictionaries**—the outer dictionary is the database, and the nested dictionaries are the records within it. Rather than a simple list of records, a dictionary-based database allows us to store and retrieve records by symbolic key:

In [58]:
```
bob = dict(name='Bob Smith', age=42, pay=30000, jpb='dev')
sue = dict(name='Sue Jones', age=45, pay=40000, jpb='hdw')
bob
```

Out[58]:
```
{'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'jpb': 'dev'}
```

In [59]:
```
db={}
db['bob'] = bob                 # reference in a dict of dicts
db['sue'] = sue
```

In [60]:
```
db['bob']['name']               # fetch bob's name
```

Out[60]:
```
'Bob Smith'
```

In [61]:
```
db['sue']['pay'] = 50000        # change sue's pay
```

In [62]:
```
db['sue']['pay']                # fetch sue's pay
```

Out[62]:
```
50000
```

In [63]:
```
import pprint
pprint.pprint(db)
```

```
{'bob': {'age': 42, 'jpb': 'dev', 'name': 'Bob Smith', 'pay': 30000},
 'sue': {'age': 45, 'jpb': 'hdw', 'name': 'Sue Jones', 'pay': 50000}}
```

If we still need to step through the database one record at a time, we can now rely on dictionary iterators. In recent Python releases, a dictionary iterator produces one key in a for loop each time through (for compatibility with earlier releases, we can also call the db.keys method explicitly in the for loop rather than saying just db , but since Python 3's keys result is a generator, the effect is roughly the same):

In [64]:
```
for key in db:
    print(key,'=>', db[key]['name'])
```

```
bob => Bob Smith
sue => Sue Jones
```

In [65]:
```
for key in db:
    print(key,'=>', db[key]['pay'])
```

```
bob => 30000
sue => 50000
```

In [66]:
```
for key in db:
    print(key,'=>', db[key]['name'])
```

```
bob => Bob Smith
sue => Sue Jones
```

To visit all records, either index by key as you go:

In [67]:
```
for key in db:
    print(db[key]['name'].split()[-1])
    db[key]['pay'] *= 1.10
```

```
Smith
Jones
```

or step through the dictionary's values to access records directly:

In [68]:
```
for record in db.values(): print(record['pay'])
```

```
33000.0
55000.00000000001
```

In [69]:
```
x = [db[key]['name'] for key in db]
x
```

Out[69]:
```
['Bob Smith', 'Sue Jones']
```

In [70]:
```
x = [rec['name'] for rec in db.values()]
x
```

Out[70]: ['Bob Smith', 'Sue Jones']

And to add a new record, simply assign it to a new key; this is just a dictionary, after all:

In [71]:
```
db['tom'] = dict(name='Tom', age=50, job=None, pay=0)
```

In [72]:
```
db['tom']
```

Out[72]: {'name': 'Tom', 'age': 50, 'job': None, 'pay': 0}

In [73]:
```
db['tom']['name']
```

Out[73]: 'Tom'

In [74]:
```
list(db.keys())
```

Out[74]: ['bob', 'sue', 'tom']

In [75]:
```
len(db)
```

Out[75]: 3

In [76]:
```
[rec['age'] for rec in db.values()]
```

Out[76]: [42, 45, 50]

In [77]:
```
[rec['name'] for rec in db.values() if rec['age'] >= 45]  # SQL-ish query
```

Out[77]: ['Sue Jones', 'Tom']