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

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

**>>> pays = map((lambda x: x[2]), people)**  # map is a generator in 3.X

**>>> list(pays)**

**[36000.0, 60000.0]**

We might try to associate names with relative positions by using the Python range built-in function:

**>>> NAME, AGE, PAY = range(3)**

**>>> bob = ['Bob Smith', 42, 10000]**

**>>> bob[NAME]**

**'Bob Smith'**

**>>> PAY, bob[PAY]**

**(2, 10000)**

Make dictionaries:

**>>> sue = {}**

**>>> sue['name'] = 'Sue Jones'**

**>>> sue['age'] = 45**

**>>> sue['pay'] = 40000**

**>>> sue['job'] = 'hdw'**

**>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')**

**>>> names = ['name', 'age', 'pay', 'job']**

**>>> values = ['Sue Jones', 45, 40000, 'hdw']**

**>>> sue = dict(zip(names, values))**

**>>> fields = ('name', 'age', 'job', 'pay')**

**>>> record = dict.fromkeys(fields, '?')**

**>>> record**

**{'job': '?', 'pay': '?', 'age': '?', 'name': '?'}**

**>>> list(map((lambda x: x['name']), people)) # generate**

**['Bob Smith', 'Sue Jones']**

**>>> [rec['name'] for rec in people if rec['age'] >= 45]**

**['Sue Jones']**

**>>> db = {}**

**>>> db['bob'] = bob**

**>>> db['sue'] = sue**

**>>> import pprint** # Python pprint pretty-printer module

**>>> pprint.pprint(db)**

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

**'sue': {'age': 45, 'job': 'hdw', 'name': 'Sue Jones', 'pay': 50000}}**

**>>> x = [db[key]['name'] for key in db]**

**>>> x = [rec['name'] for rec in db.values()]**

**>>> db['tom'] = dict(name='Tom', age=50, job=None, pay=0)**

**>>> [rec['name'] for rec in db.values() if rec['age'] >= 45]**

**['Sue Jones', 'Tom']**

The pickle module translates an in-memory Python object into a serialized byte stream—a string of bytes that can be written to any file-like object.

The pickle module also knows how to reconstruct the original object in memory, given the serialized byte stream.

As of Python 3, all protocols use bytes objects to represent pickled data, which in turn requires pickle files to be opened in binary mode for all protocols.

|  |  |
| --- | --- |
| **Dbfile:**  bob  job=>'dev'  pay=>30000  age=>42  name=>'Bob Smith'  endrec.  sue  job=>'hdw'  pay=>40000  age=>45  name=>'Sue Jones'  endrec.  tom  job=>None  pay=>0  age=>50  name=>'Tom'  endrec.  enddb. | **def loadDbase(dbfilename=dbfilename):**  **dbfile = open(dbfilename)**  **import sys**  **sys.stdin = dbfile**  **db = {}**  **key = input()**  **while key != ENDDB:**  **rec = {}**  **field = input()**  **while field != ENDREC:**  **name, value = field.split(RECSEP)**  **rec[name] = eval(value)**  **field = input()**  **db[key] = rec**  **key = input()**  **return db** |
| **from initdata import db**  **import pickle**  **dbfile = open('people-pickle', 'wb')**  **pickle.dump(db, dbfile)**  **dbfile.close()** |
| **import pickle**  **dbfile = open('people-pickle', 'rb')**  **db = pickle.load(dbfile)**  **dbfile.close()**  **db['sue']['pay'] \*= 1.10**  **db['tom']['name'] = 'Tom Tom'**  **dbfile = open('people-pickle', 'wb')**  **pickle.dump(db, dbfile)**  **dbfile.close()** |
| **from initdata import tom**  **import shelve**  **db = shelve.open('people-shelve')**  **sue = db['sue']** # fetch sue  **sue['pay'] \*= 1.50** # update sue  **db['sue'] = sue** # add a new record  **db['tom'] = tom**  **db.close()** |

|  |  |
| --- | --- |
| class Person | **class Person:**  **def \_\_init\_\_(self, name, age, pay=0, job=None):**  **self.name = name**  **self.age = age**  **self.pay = pay**  **self.job = job**  **def lastName(self):**  **return self.name.split()[-1]**  **def giveRaise(self, percent):**  **self.pay \*= (1.0 + percent)** |
| class Manager  inherits from  Person | **class Manager(Person):**  **def giveRaise(self, percent, bonus=0.1):**  **self.pay \*= (1.0 + percent + bonus)** |
| reduce code redundance by  calling back | **class Manager(Person):**  **def giveRaise(self, percent, bonus=0.1):**  **Person.giveRaise(self, percent + bonus)** |

In general, the following are equivalent, and both forms may be used explicitly:

**instance.method(arg1, arg2)**

**class.method(instance, arg1, arg2)**

operator overloading methods:

**class Person:**

**def \_\_str\_\_(self):**

**return '<%s => %s>' % (self.\_\_class\_\_.\_\_name\_\_, self.name)**

**tom = Manager('Tom Jones', 50)**

**print(tom) # prints: <Manager => Tom Jones>**

Here **\_\_class\_\_** gives the lowest class from which self was made, even though **\_\_str\_\_** may be inherited.

An explicit constructor for managers:

**class Manager(Person):**

**def \_\_init\_\_(self, name, age, pay):**

**Person.\_\_init\_\_(self, name, age, pay, 'manager')**

Now when a manager is created, its job is filled in automatically.

When instances are shelved or pickled, the underlying pickling system records both instance attributes and enough information to locate their classes automatically when they are later fetched (the class’s module simply has to be on the module search path when an instance is loaded).

# interactive queries

**import shelve**

**fieldnames = ('name', 'age', 'job', 'pay')**

**maxfield = max(len(f) for f in fieldnames)**

**db = shelve.open('class-shelve')**

**while True:**

**key = input('\nKey? => ')**

**if not key: break**

**try:**

**record = db[key]**

**except:**

**print('No such key "%s"!' % key)**

**else:**

**for field in fieldnames:**

**print(field.ljust(maxfield), '=>', getattr(record, field))**