**>>> 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))**

Implements a GUI with a button that runs the reply function each time it is

pressed.

**from tkinter import \***

**from tkinter.messagebox import showinfo**

**def reply():**

**showinfo(title='popup', message='Button pressed!')**

**window = Tk()**

**button = Button(window, text='press', command=reply)**

**button.pack()**

**window.mainloop()**

Using OOP for GUIs:

**from tkinter import \***

**from tkinter.messagebox import showinfo**

**class MyGui(Frame):**

**def \_\_init\_\_(self, parent=None):**

**Frame.\_\_init\_\_(self, parent)**

**button = Button(self, text='press', command=self.reply)**

**button.pack()**

**def reply(self):**

**showinfo(title='popup', message='Button pressed!')**

**if \_\_name\_\_ == '\_\_main\_\_':**

**window = MyGui()**

**window.pack()**

**window.mainloop()**

MyGui is a subclass of Frame, it automatically becomes an **attachable** component—i.e., we can add all of the widgets this class creates, as a package, to any other GUI, just by attaching this Frame to the GUI.

**from tkinter import \***

**from tkinter102 import MyGui**

**# main app window**

**mainwin = Tk()**

**Label(mainwin, text=\_\_name\_\_).pack()**

**# popup window**

**popup = Toplevel()**

**Label(popup, text='Attach').pack(side=LEFT)**

**MyGui(popup).pack(side=RIGHT) # attach my frame**

**mainwin.mainloop()**

python cgi小记：

sudo tail -f /var/log/apache2/error.log 查看动态错误日志

默认cgi-bin路径是'/usr/lib/cgi-bin/',由于peoplecgi.py还要import同目录下的person.py,并且需要打开class-shelve.db,为了图方便,修改一下cgi-bin的路径,使其指向目前源代码所在目录：sudo vim /etc/apache2/conf-enabled/serve-cgi-bin.conf

找到如下的两行：

ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/

<Directory "/usr/lib/cgi-bin">

作出相应修改。

sudo services apache2 restart

出现权限错误（参考https://wiki.python.org/moin/CgiScripts）,修改peoplecgi.py等相关文件的权限为755,浏览器仍然显示500错误,查看日志中有如下错误：

[client ::1:52573] AH01215: File "/home/simon/Learning/python/PP4E/peoplecgi.py", line 75, in <module>: /home/simon/Learning/python/PP4E/peoplecgi.py……

[client ::1:52573] AH01215: db = shelve.open(shelvename): ……

[client ::1:52573] AH01215: \_dbm.error: [Errno 13] Permission denied: ……

判断是在打开class-shelve.db时遇到的权限问题,增加其它用户写入权限：

chmod o+w class-shelve.db

终于正常。

A %(key)code replacement target fetches a value by key from a dictionary:

**>>> D = {'say': 5, 'get': 'shrubbery'}**

**>>> D['say']**

**5**

**>>> S = '%(say)s => %(get)s' % D**

**>>> S**

**'5 => shrubbery'**

**>>> from person import Person**

**>>> bob = Person('Bob', 35)**

**>>> '%(name)s, %(age)s' % bob.\_\_dict\_\_**

**'Bob, 35'**

**>>> line = 'aaa\nbbb\nccc\n'**

**>>> line.split('\n')**

**['aaa', 'bbb', 'ccc', '']**

**>>> line.splitlines()**

**['aaa', 'bbb', 'ccc']**

**>>> mystr = 'xxxSPAMxxx'**

**>>> mystr.find('SPAM') # return first offset**

**3**

**>>> mystr.find('Ni') # when not found**

**-1**

**>>> mystr = 'xxaaxxaa'**

**>>> mystr.replace('aa', 'SPAM') # global replacement**

**'xxSPAMxxSPAM'**

**>>> mystr = '\t Ni\n' # remove whitespace**

**>>> mystr.strip()**

**'Ni'**

**>>> mystr.rstrip() # same, but just on right side**

**'\t Ni'**

**>>> mystr = 'SHRUBBERY'**

**>>> mystr.lower()**

**'shrubbery'**

**>>> mystr.isalpha()**

**True**

**>>> mystr.isdigit()**

**False**

**>>> import string**

**>>> string.ascii\_lowercase**

**'abcdefghijklmnopqrstuvwxyz'**

**>>> string.whitespace**

**' \t\n\r\x0b\x0c'**

**>>> mystr = 'aaa,bbb,ccc'**

**>>> mystr.split(',')**

**['aaa', 'bbb', 'ccc'] # split into substrings list**

**>>> mystr = 'a b\nc\nd'**

**>>> mystr.split()**

**['a', 'b', 'c', 'd'] # default delimiter: whitespace**

**>>> delim = 'NI'**

**>>> delim.join(['aaa', 'bbb', 'ccc'])**

**'aaaNIbbbNIccc' # join substrings list**

**>>> ' '.join(['A', 'dead', 'parrot'])**

**'A dead parrot'**

**>>> chars = list('Lorreta') # convert to characters list**

**>>> chars**

**['L', 'o', 'r', 'r', 'e', 't', 'a']**

**>>> chars.append('!')**

**>>> ''.join(chars) # to string: empty delimiter**

**'Lorreta!'**

**>>> int("42"), eval("42")**

**(42, 42) # string to int conversions**

**>>> str(42), repr(42)**

**('42', '42') # int to string conversions**

**>>> ("%d" % 42), '{:d}'.format(42)**

**('42', '42') # via formatting expression, method**

**>>> "42" + str(1), int("42") + 1**

**('421', 43) # concatenation, addition**

bytes —— a sequence of short integers for representing 8-bit binary data

bytearray —— a mutable variant of bytes.

Files in Python 3.X follow a similar dichotomy, using str in text mode (which also handles Unicode encodings and line-end conversions) and bytes in binary mode (which transfers bytes to and from files

unchanged).

open('file').read() # read entire file into string

open('file').read(N) # next N bytes into string

open('file').readlines() # entire file into line strings list

open('file').readline() # next line, through '\n'

Every Python module has a built-in \_\_name\_\_ variable that Python sets to the \_\_main\_\_ string only when the file is run as a program, not when it’s imported as a library.

Changes to sys.path are retained only until the

Python process ends, and they must be remade every time you start a new Python

program or session.

sys.exc\_info function returns a tuple with the latest exception’s type, value, and traceback object.

|  |  |
| --- | --- |
| **Tasks** | **Tools** |
| Shell variables | os.environ |
| Running programs | os.system, os.popen, os.execv, os.spawnv |
| Spawning processes | os.fork, os.pipe, os.waitpid, os.kill |
| Descriptor files, locks | os.open, os.read, os.write |
| File processing | os.remove, os.rename, os.mkfifo, os.mkdir, os.rmdir |
| Administrative tools | os.getcwd, os.chdir, os.chmod, os.getpid, os.listdir, os.access |
| Portability tools | os.sep, os.pathsep, os.curdir, os.path.split, os.path.join |
| Pathname tools | os.path.exists('path'), os.path.isdir('path'), os.path.getsize('path') |

**>>> os.pathsep, os.sep, os.pardir, os.curdir, os.linesep**

**(':', '/', '..', '.', '\n')**

For instance, a call of the form dir path.split(**os.sep**) will correctly split platform-specific directory names into components.

**>>> os.path.split('~/Learning/python/PP4E/System/more.py')**

**('~/Learning/python/PP4E/System', 'more.py')**

**>>> os.path.join('~/Learning/python/PP4E/System', 'more.py')**

**'~/Learning/python/PP4E/System/more.py'**

**>>> name = r'~/Learning/python/PP4E/System/more.py'**

**>>> os.path.dirname(name), os.path.basename(name)**

**('~/Learning/python/PP4E/System', 'more.py')**

**>>> os.path.splitext('~/Learning/python/PP4E/System/more.py')**

**('~/Learning/python/PP4E/System/more', '.py')**

**os.system**

**Runs a shell command from a Python script**

**os.popen**

**Runs a shell command and connects to its input or output streams**

**>>> os.system('uname -r')**

**3.14-2-amd64**

**0**

The 0 s at the end of the first two commands here are just the return values of the system call itself (its exit status; zero generally means success).

The **os.system** call simply runs a shell command line, but **os.popen** also connects to the standard input or output streams of the command; we get back a file-like object connected to the command’s output by default (if we pass a **w** mode flag to popen , we connect to the command’s input stream instead).

>>> open('helloshell.py').read()

"#!/usr/bin/env python3\nprint('The Meaning of Life')\n"

>>> text = os.popen('cat helloshell.py').read()

>>> text

"#!/usr/bin/env python3\nprint('The Meaning of Life')\n"

>>> listing = os.popen('ls').readlines()

>>> listing

['helloshell.py\n', 'more.py\n', '\_\_pycache\_\_\n']

>>> os.system('python3 helloshell.py')

The Meaning of Life

0

>>> output = os.popen('python helloshell.py').read()

>>> output

'The Meaning of Life\n'

>>> import subprocess

>>> subprocess.call('python helloshell.py')

The Meaning of Life

0

>>> subprocess.call('cat helloshell.py', shell=True)

#!/usr/bin/env python3

print('The Meaning of Life')

0

On Unix-like platforms, when **shell** is **False** (its default), the program command line is run directly by **os.execvp**. If this argument is

True, the command-line string is run through a shell instead, and you can specify the shell to use with additional arguments.

**>>> pipe = subprocess.Popen(['python3', 'helloshell.py'], stdout=subprocess.PIPE)**

**>>> pipe.communicate()**

**(b'The Meaning of Life\n', None)**

**>>> pipe.returncode**

**0**

We connect the stdout stream to a pipe, and communicate to run the command

to completion and receive its standard output and error streams’text.

**>>> pipe = subprocess.Popen(['python3', 'helloshell.py'], stdout=subprocess.PIPE)**

**>>> pipe.stdout.read()**

**b'The Meaning of Life\n'**

**>>> pipe.wait()**

**0**

>>> import os

>>> os.popen('python3 helloshell.py').read()

'The Meaning of Life\n'

Because the **print** and **input** built-in functions are really nothing more than user-friendly interfaces to the standard output and input streams, they are similar to using **stdout** and **stdin** in **sys** directly:

**>>> print('hello stdout world')**

**hello stdout world**

**>>> sys.stdout.write('hello stdout world' + '\n')**

**hello stdout world**

**19**

**>>> input('hello stdin world>')**

**hello stdin world>spam**

**'spam'**

**>>> print('hello stdin world>'); sys.stdin.readline()[:-1]**

**hello stdin world>**

**eggs**

**'eggs'**