# Annexes

## A. How to use a raw socket

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recv(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

## B. How to convert a 32-bit packed IP address to its standard dotted string

`inet_ntoa` (*packed_ip*)

Convert a 32-bit packed IP address (a string four characters in length) to its standard dotted-quad string representation (e.g. '123.45.67.89').

If the string passed to this function is not exactly 4 bytes in length, `socket.error` will be raised.

## C. How to decode network frames

`struct.unpack`(*fmt*, *string*)¶

Unpack the string (presumably packed by `pack(fmt, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (`len(string)` must equal `calcsize(fmt)`).

| Format | C Type | Python |
|---|---|---|
| x | Pad byte | No value |
| c | Char | String of length 1 |
| b | Signed char | Integer |
| B | Unsigned char | Integer |
| ? | _Bool | Bool |
| h | Short | Integer |
| H | Unsigned short | Integer |
| i | Int | Integer or long |
| l | Long | Integer |
| L | Unsigned long | Long |
| q | Long long | Long |
| Q | Unsigned long long | long |
| f | float | float |
| d | double | float |
| s | char[] | string |
| p | char[] | string |
| P | Void * | long |

Unpacking data, you have to care about the endianness, or byte order.
They are mainly 2 types: big-endian and little-endian.
With the exemple of storing 0xABCD in memory, with increasing address from right to left. Using 8 bit atomic words:

| | | |
|---|---|---|
| Big-Endian | AB | CD |
| Little-Endian | CD | AB |

| Character | Byte order | Size and alignment |
|---|---|---|
| @ | Native | Native |
| = | Native | Standard |
| < | Little-endian | Standard |
| > | Big-endian | Standard |
| ! | Network = big-endian | Standard |

## D. How to use a dict

A *mapping* object maps *hashable* values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built in list, set, and tuple classes, and the collections module.)

A dictionary's keys are *almost* arbitrary values. Values that are not *hashable*, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}, or by the dict constructor.

*class* dict([*arg*])¶

Return a new dictionary initialized from an optional positional argument or from a set of keyword arguments. If no arguments are given, return a new empty dictionary. If the positional argument *arg* is a mapping object, return a dictionary mapping the same keys to the same values as does the mapping object. Otherwise the positional argument must be a sequence, a container that supports iteration, or an iterator object. The elements of the argument must each also be of one of those kinds, and each must in turn contain exactly two objects. The first is used as a key in the new dictionary, and the second as the key's value. If a given key is seen more than once, the last value associated with it is retained in the new dictionary.

If keyword arguments are given, the keywords themselves with their associated values are added as items to the dictionary. If a key is specified both in the positional argument and as a keyword argument, the value associated with the keyword is retained in the dictionary. For example, these all return a dictionary equal to {"one": 2, "two": 3}:

- dict(one=2, two=3)
- dict({'one': 2, 'two': 3})
- dict(zip(('one', 'two'), (2, 3)))
- dict([['two', 3], ['one', 2]])

The first example only works for keys that are valid Python identifiers; the others work with any valid keys.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

len(d)
Return the number of items in the dictionary *d*.
d[key]

Return the item of *d* with key *key*. Raises a KeyError if *key* is not in the map.

d[key] = value
Set d[key] to *value*.

del d[key]
Remove d[key] from *d*. Raises a KeyError if *key* is not in the map.

key in d

Return True if *d* has a key *key*, else False.

key not in d

Equivalent to not key in d.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for iterkeys().

clear()¶
Remove all items from the dictionary.

copy()¶
Return a shallow copy of the dictionary.

fromkeys(*seq*[, *value*])¶

Create a new dictionary with keys from *seq* and values set to *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to None.

get(*key*[, *default*])¶
Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to None, so that this method never raises a KeyError.

has_key(*key*)¶
Test for the presence of *key* in the dictionary. has_key() is deprecated in favor of key in d.

items()¶

Return a copy of the dictionary's list of (key, value) pairs.

iteritems()¶

Return an iterator over the dictionary's (key, value) pairs. See the note for dict.items().

Using iteritems() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

iterkeys()¶

Return an iterator over the dictionary's keys. See the note for dict.items().

Using iterkeys() while adding or deleting entries in the dictionary may raise a RuntimeError

or fail to iterate over all entries.

itervalues()¶

Return an iterator over the dictionary's values. See the note for dict.items().

Using itervalues() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

keys()¶

Return a copy of the dictionary's list of keys. See the note for dict.items().

pop(*key*[, *default*])¶

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a KeyError is raised.

## E. How to use binary files to store data in a flatten format

```
import os

outFd= os.open("test.bin", os.O_RDWR | os.O_CREAT)

os.write(outFd,"\xFe\xed\xba\xbe")

os.close(outFd)
```

hexdump test.bin
0000000 edfe beba
0000004

Another way:

```
dataFile = open("test.bin", “w”)

dataFile.write("\xFe\xed\xba\xbe")

dataFile.close()
```