guess_extension(*type*, *strict=True*)

Similar to the `guess_extension()` function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)

Similar to the `guess_type()` function, using the tables stored as part of the object.

guess_all_extensions(*type*, *strict=True*)

Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.

read(*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses `readfp()` to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)

Load MIME type information from the Windows registry.

Availability: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

New in version 3.2.

19.5 `base64` — Base16, Base32, Base64, Base85 Data Encodings

Source code: Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. It provides encoding and decoding functions for the encodings specified in **RFC 4648**, which defines the Base16, Base32, and Base64 algorithms, and for the de-facto standard Ascii85 and Base85 encodings.

The **RFC 4648** encodings are suitable for encoding binary data so that it can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the **uuencode** program.

There are two interfaces provided by this module. The modern interface supports encoding *bytes-like objects* to ASCII `bytes`, and decoding *bytes-like objects* or strings containing ASCII to `bytes`. Both base-64 alphabets defined in **RFC 4648** (normal, and URL- and filesystem-safe) are supported.

The legacy interface does not support decoding from strings, but it does provide functions for encoding and decoding to and from *file objects*. It only supports the Base64 standard alphabet, and it adds newlines every 76 characters as per **RFC 2045**. Note that if you are looking for **RFC 2045** support you probably want to be looking at the `email` package instead.

Changed in version 3.3: ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface.

Changed in version 3.4: Any *bytes-like objects* are now accepted by all encoding and decoding functions in this module. Ascii85/Base85 support added.

The modern interface provides:

`base64.`**`b64encode`**`(`*s*, *altchars=None*`)`

Encode the *bytes-like object* *s* using Base64 and return the encoded `bytes`.

Optional *altchars* must be a *bytes-like object* of length 2 which specifies an alternative alphabet for the `+` and `/` characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

May assert or raise a `ValueError` if the length of *altchars* is not 2. Raises a `TypeError` if *altchars* is not a *bytes-like object*.

`base64.`**`b64decode`**`(`*s*, *altchars=None*, *validate=False*`)`

Decode the Base64 encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`.

Optional *altchars* must be a *bytes-like object* or ASCII string of length 2 which specifies the alternative alphabet used instead of the `+` and `/` characters.

A `binascii.Error` exception is raised if *s* is incorrectly padded.

If *validate* is `False` (the default), characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check. If *validate* is `True`, these non-alphabet characters in the input result in a `binascii.Error`.

For more information about the strict base64 check, see `binascii.a2b_base64()`

May assert or raise a `ValueError` if the length of *altchars* is not 2.

`base64.`**`standard_b64encode`**`(`*s*`)`

Encode *bytes-like object* *s* using the standard Base64 alphabet and return the encoded `bytes`.

`base64.`**`standard_b64decode`**`(`*s*`)`

Decode *bytes-like object* or ASCII string *s* using the standard Base64 alphabet and return the decoded `bytes`.

`base64.`**`urlsafe_b64encode`**`(`*s*`)`

Encode *bytes-like object* *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the encoded `bytes`. The result can still contain `=`.

`base64.`**`urlsafe_b64decode`**`(`*s*`)`

Decode *bytes-like object* or ASCII string *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the decoded `bytes`.

`base64.`**`b32encode`**`(`*s*`)`

Encode the *bytes-like object* *s* using Base32 and return the encoded `bytes`.

`base64.`**`b32decode`**`(`*s*, *casefold=False*, *map01=None*`)`

Decode the Base32 encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

RFC 4648 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not `None`, specifies which letter the digit 1 should be mapped to (when *map01* is not `None`, the digit 0 is always mapped to the letter O). For security purposes the default is `None`, so that 0 and 1 are not allowed in the input.

A `binascii.Error` is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

`base64.`**`b32hexencode`**`(`*s*`)`

Similar to `b32encode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

New in version 3.10.

`base64.`**`b32hexdecode`**`(`*s*, *casefold=False*`)`

Similar to `b32decode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

This version does not allow the digit 0 (zero) to the letter O (oh) and digit 1 (one) to either the letter I (eye) or letter L (el) mappings, all these characters are included in the Extended Hex Alphabet and are not interchangeable.

New in version 3.10.

`base64.`**`b16encode`**`(`*s*`)`

Encode the *bytes-like object* *s* using Base16 and return the encoded `bytes`.

`base64.`**`b16decode`**`(`*s*, *casefold=False*`)`

Decode the Base16 encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

A `binascii.Error` is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

`base64.`**`a85encode`**`(`*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*`)`

Encode the *bytes-like object* *b* using Ascii85 and return the encoded `bytes`.

foldspaces is an optional flag that uses the special short sequence 'y' instead of 4 consecutive spaces (ASCII 0x20) as supported by 'btoa'. This feature is not supported by the "standard" Ascii85 encoding.

wrapcol controls whether the output should have newline (`b'\n'`) characters added to it. If this is non-zero, each output line will be at most this many characters long.

pad controls whether the input is padded to a multiple of 4 before encoding. Note that the `btoa` implementation always pads.

adobe controls whether the encoded byte sequence is framed with `<~` and `~>`, which is used by the Adobe implementation.

New in version 3.4.

`base64.`**`a85decode`**`(`*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*`)`

Decode the Ascii85 encoded *bytes-like object* or ASCII string *b* and return the decoded `bytes`.

foldspaces is a flag that specifies whether the 'y' short sequence should be accepted as shorthand for 4 consecutive spaces (ASCII 0x20). This feature is not supported by the "standard" Ascii85 encoding.

adobe controls whether the input sequence is in Adobe Ascii85 format (i.e. is framed with <~ and ~>).

ignorechars should be a *bytes-like object* or ASCII string containing characters to ignore from the input. This should only contain whitespace characters, and by default contains all whitespace characters in ASCII.

New in version 3.4.

`base64.`**`b85encode`**`(`*b*, *pad=False*`)`

Encode the *bytes-like object* *b* using base85 (as used in e.g. git-style binary diffs) and return the encoded `bytes`.

If *pad* is true, the input is padded with `b'\0'` so its length is a multiple of 4 bytes before encoding.

New in version 3.4.

`base64.`**`b85decode`**`(`*b*`)`

Decode the base85-encoded *bytes-like object* or ASCII string *b* and return the decoded `bytes`. Padding is implicitly removed, if necessary.

New in version 3.4.

The legacy interface:

`base64.`**`decode`**`(`*input*, *output*`)`

Decode the contents of the binary *input* file and write the resulting binary data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.readline()` returns an empty bytes object.

base64.**decodebytes**(*s*)

Decode the *bytes-like object* *s*, which must contain one or more lines of base64 encoded data, and return the decoded `bytes`.

New in version 3.1.

base64.**encode**(*input*, *output*)

Encode the contents of the binary *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.read()` returns an empty bytes object. `encode()` inserts a newline character (`b'\n'`) after every 76 bytes of the output, as well as ensuring that the output always ends with a newline, as per **RFC 2045** (MIME).

base64.**encodebytes**(*s*)

Encode the *bytes-like object* *s*, which can contain arbitrary binary data, and return `bytes` containing the base64-encoded data, with newlines (`b'\n'`) inserted after every 76 bytes of output, and ensuring that there is a trailing newline, as per **RFC 2045** (MIME).

New in version 3.1.

An example usage of the module:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

19.5.1 Security Considerations

A new security considerations section was added to **RFC 4648** (section 12); it's recommended to review the security section for any code deployed to production.

See also:

Module `binascii` Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Form
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

19.6 `binascii` — Convert between binary and ASCII

The `binascii` module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like `uu` or `base64` instead. The `binascii` module contains low-level functions written in C for greater speed that are used by the higher-level modules.

Note: `a2b_*` functions accept Unicode strings containing only ASCII characters. Other functions only accept *bytes-like objects* (such as `bytes`, `bytearray` and other objects that support the buffer protocol).

Changed in version 3.3: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

The `binascii` module defines the following functions: