Appendix A: String Functions

capitalize(...)  
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.

center(...)  
    S.center(width[, fillchar]) -> str

    Return S centered in a string of length width. Padding is
    done using the specified fill character (default is a space)

count(...)  
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end]. Optional arguments start and end are
    interpreted as in slice notation.

code(...)  
    S.encode(encoding='utf-8', errors='strict') -> bytes

    Encode S using the codec registered for encoding. Default encoding
    is 'utf-8'. errors may be given to set a different error
    handling scheme. Default is 'strict' meaning that encoding errors raise
    a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
    'xmlcharrefreplace' as well as any other name registered with
    codecs.register_error that can handle UnicodeEncodeErrors.

denswith(...)
S.endswith(suffix[, start[, end]]) -> bool

Return True if S ends with the specified suffix, False otherwise.
With optional start, test S beginning at that position.
With optional end, stop comparing S at that position.
suffix can also be a tuple of strings to try.

expandtabs(...)
S.expandtabs([tabsize]) -> str

Return a copy of S where all tab characters are expanded using spaces.
If tabsize is not given, a tab size of 8 characters is assumed.

find(...)
S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within S[start:end]. Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(...)
S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs.
The substitutions are identified by braces ('{' and '}').

format_map(...)
S.format_map(mapping) -> str

Return a formatted version of S, using substitutions from mapping.
The substitutions are identified by braces (''{'}').

index(...)
S.index(sub[, start[, end]]) -> int

Like S.find() but raise ValueError when the substring is not found.

isalnum(...)
S.isalnum() -> bool

Return True if all characters in S are alphanumeric
and there is at least one character in S, False otherwise.

isalpha(...)
S.isalpha() -> bool
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal(...)  
S.isdecimal() -> bool

Return True if there are only decimal characters in S, False otherwise.

isdigit(...)  
S.isdigit() -> bool

Return True if all characters in S are digits and there is at least one character in S, False otherwise.

isidentifier(...)  
S.isidentifier() -> bool

Return True if S is a valid identifier according to the language definition.

islower(...)  
S.islower() -> bool

Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric(...)  
S.isnumeric() -> bool

Return True if there are only numeric characters in S, False otherwise.

isprintable(...)  
S.isprintable() -> bool

Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.

isspace(...)  
S.isspace() -> bool

Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle(...)
S.istitle() -> bool

Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper(...)  
S.isupper() -> bool

Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join(...)  
S.join(iterable) -> str

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust(...)  
S.ljust(width[, fillchar]) -> str

Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower(...)  
S.lower() -> str

Return a copy of the string S converted to lowercase.

lstrip(...)  
S.lstrip([chars]) -> str

Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead.

partition(...)  
S.partition(sep) -> (head, sep, tail)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace(...)  
S.replace(old, new[, count]) -> str

Return a copy of S with all occurrences of substring
old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(...)  
S.rfind(sub[, start[, end]]) -> int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(...)  
S.rindex(sub[, start[, end]]) -> int

Like S.rfind() but raise ValueError when the substring is not found.

rjust(...)  
S.rjust(width[, fillchar]) -> str

Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space).

rpartition(...)  
S.rpartition(sep) -> (head, sep, tail)

Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit(...)  
S.rsplit([sep[, maxsplit]]) -> list of strings

Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip(...)  
S.rstrip([chars]) -> str

Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

split(...)  
S.split([sep[, maxsplit]]) -> list of strings
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

**splitlines(...)**

S.splitlines([keepends]) -> list of strings

Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

**startswith(...)**

S.startswith(prefix[, start[, end]]) -> bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

**strip(...)**

S.strip([chars]) -> str

Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

**swapcase(...)**

S.swapcase() -> str

Return a copy of S with uppercase characters converted to lowercase and vice versa.

**title(...)**

S.title() -> str

Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

**translate(...)**

S.translate(table) -> str

Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
Unmapped characters are left untouched. Characters mapped to None are deleted.

`upper(...)`

```python
S.upper() -> str
```

Return a copy of S converted to uppercase.

`zfill(...)`

```python
S.zfill(width) -> str
```

Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

## Appendix B: String Formatter

### Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax).

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name ::= arg_name[."" attribute_name | "{" element_index "]")*      
arg_name ::= [identifier | integer]
attribute_name ::= identifier
element_index ::= integer | index_string
index_string ::= <any source character except "]"> +
conversion ::= "r" | "s" | "a"
format_spec ::= <described in the next section>
```

In less formal terms, the replacement field can start with a `field_name` that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The `field_name` is optionally followed by a `conversion` field, which is preceded by an exclamation point `!`, and a `format_spec`, which is preceded by a colon `:`. These specify a non-default format for the replacement value.

See also the Format Specification Mini-Language section.

The `field_name` itself begins with an `arg_name` that is either a number or a keyword. If it’s a number, it refers to a positional argument, and if it’s a keyword, it refers to a named keyword argument. If the numerical `arg_names` in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because `arg_name` is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The `arg_name` can be followed by any number of index or attribute expressions.
An expression of the form `.name` selects the named attribute using `getattr()`, while an expression of the form `[index]` does an index lookup using `__getitem__()`.  

*Changed in version 3.1:* The positional argument specifiers can be omitted, so `{0} {1}` is equivalent to `{0} {1}`.

Some simple format string examples:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}" # Implicitly references the first positional argument
"From {} to {}" # Same as "From {0} to {1}"
"My quest is {name}" # References keyword argument 'name'
"Weight in tons {0.weight}" # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}" # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the `__format__()` method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

Three conversion flags are currently supported: `!s` which calls `str()` on the value, `!r` which calls `repr()` and `!a` which calls `ascii()`.

Some examples:

```
"Harold's a clever {0!s}" # Calls str() on the argument first
"Bring out the holy {name!r}" # Calls repr() on the argument first
"More {!a}" # Calls ascii() on the argument first
```

The *format_spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the `format_spec`.

Most built-in types support a common formatting mini-language, which is described in the next section.

A `format_spec` field can also include nested replacement fields within it. These nested replacement fields can contain only a field name; conversion flags and format specifications are not allowed. The replacement fields within the `format_spec` are substituted before the `format_spec` string is interpreted. This allows the formatting of a value to be dynamically specified.

### Format Specification Mini-Language

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see *Format String Syntax*). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string ("") produces the same result as if you had called `str()` on the value. A non-empty format string typically modifies the result.

The general form of a *standard format specifier* is:
The fill character can be any character other than ‘{‘ or ‘}’. The presence of a fill character is signaled by the character following it, which must be one of the alignment options. If the second character of format_spec is not a valid alignment option, then it is assumed that both the fill character and the alignment option are absent.

The meaning of the various alignment options is as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Forces the field to be left-aligned within the available space (this is the default for most objects).</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Forces the field to be right-aligned within the available space (this is the default for numbers).</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+00000120’. This alignment option is only valid for numeric types.</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Forces the field to be centered within the available space.</td>
</tr>
</tbody>
</table>

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The sign option is only valid for number types, and can be one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>indicates that a sign should be used for both positive as well as negative numbers.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>indicates that a sign should be used only for negative numbers (this is the default behavior).</td>
</tr>
<tr>
<td><code>space</code></td>
<td>indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.</td>
</tr>
</tbody>
</table>

The `#` option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective `0b`, `0o`, or `0x` to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `g` and `G` conversions, trailing zeros are not removed from the result.

The `,` option signals the use of a comma for a thousands separator. For a locale aware separator, use the `n` integer presentation type instead.

*Changed in version 3.1:* Added the `,` option (see also PEP 378).

width is a decimal integer defining the minimum field width. If not specified, then the field width will
be determined by the content.

Preceding the width field by a zero (‘0’) character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of ‘0’ with an alignment type of ‘=’.

The precision is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with ‘f’ and ‘F’, or before and after the decimal point for a floating point value formatted with ‘g’ or ‘G’. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The precision is not allowed for integer values.

Finally, the type determines how the data should be presented.

The available string presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘s’</td>
<td>String format. This is the default type for strings and may be omitted.</td>
</tr>
<tr>
<td>None</td>
<td>The same as ‘s’.</td>
</tr>
</tbody>
</table>

The available integer presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘b’</td>
<td>Binary format. Outputs the number in base 2.</td>
</tr>
<tr>
<td>‘c’</td>
<td>Character. Converts the integer to the corresponding unicode character before printing.</td>
</tr>
<tr>
<td>‘d’</td>
<td>Decimal Integer. Outputs the number in base 10.</td>
</tr>
<tr>
<td>‘o’</td>
<td>Octal format. Outputs the number in base 8.</td>
</tr>
<tr>
<td>‘x’</td>
<td>Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.</td>
</tr>
<tr>
<td>‘X’</td>
<td>Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.</td>
</tr>
<tr>
<td>‘n’</td>
<td>Number. This is the same as ‘d’, except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td>None</td>
<td>The same as ‘d’.</td>
</tr>
</tbody>
</table>

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except ‘n’ and None). When doing so, float() is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘e’</td>
<td>Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6.</td>
</tr>
<tr>
<td>‘E’</td>
<td>Exponent notation. Same as ‘e’ except it uses an upper case ‘E’ as the separator character.</td>
</tr>
<tr>
<td>‘f’</td>
<td>Fixed point. Displays the number as a fixed-point number. The default precision is 6.</td>
</tr>
<tr>
<td>‘F’</td>
<td>Fixed point. Same as ‘f’, but converts nan to NAN and inf to INF.</td>
</tr>
</tbody>
</table>
### Type | Meaning
---|---
'g' | General format. For a given precision \(p \geq 1\), this rounds the number to \(p\) significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude.

The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision \(p-1\) would have exponent \(\text{exp}\). Then if \(-4 \leq \exp < p\), the number is formatted with presentation type 'f' and precision \(p-1-\exp\). Otherwise, the number is formatted with presentation type 'e' and precision \(p-1\). In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it.

Positive and negative infinity, positive and negative zero, and nans, are formatted as \(\text{inf, -inf, 0, -0 and nan}\) respectively, regardless of the precision.

A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.

'G' | General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.

'n' | Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.

's' | Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

None | Similar to 'g', except with at least one digit past the decimal point and a default precision of 12. This is intended to match \(\text{str()}\), except you can add the other format modifiers.

### Appendix C: List Functions

append(...)  
L.append(object) -- append object to end

count(...)  
L.count(value) -> integer -- return number of occurrences of value

extend(...)  
L.extend(iterable) -- extend list by appending elements from the iterable

index(...)  
L.index(value, [start, [stop]]) -> integer -- return first index of value.

Raises ValueError if the value is not present.
insert(...)  
L.insert(index, object) -- insert object before index

pop(...)  
L.pop([index]) -> item -- remove and return item at index (default last).  
    Raises IndexError if list is empty or index is out of range.

remove(...)  
L.remove(value) -- remove first occurrence of value.  
    Raises ValueError if the value is not present.

reverse(...)  
L.reverse() -- reverse *IN PLACE*

sort(...)  
L.sort(key=None, reverse=False) -- stable sort *IN PLACE*

---

**Appendix D: Tuple Functions**

count(...)  
T.count(value) -> integer -- return number of occurrences of value

index(...)  
T.index(value, [start, [stop]]) -> integer -- return first index of value.  
    Raises ValueError if the value is not present.

---

**Appendix E: Set Functions**

add(...)  
Add an element to a set.  
    This has no effect if the element is already present.

clear(...)  
Remove all elements from this set.

copy(...)  
Return a shallow copy of a set.

difference(...)  
Return the difference of two or more sets as a new set.  
    (i.e. all elements that are in this set but not the others.)
discard(...)  
Remove an element from a set if it is a member. If the element is not a member, do nothing.

intersection(...)  
Return the intersection of two sets as a new set.  
(i.e. all elements that are in both sets.)

isdisjoint(...)  
Return True if two sets have a null intersection.

issubset(...)  
Report whether another set contains this set.

issuperset(...)  
Report whether this set contains another set.

pop(...)  
Remove and return an arbitrary set element.  
Raises KeyError if the set is empty.

remove(...)  
Remove an element from a set; it must be a member.  
If the element is not a member, raise a KeyError.

symmetric_difference(...)  
Return the symmetric difference of two sets as a new set.  
(i.e. all elements that are in exactly one of the sets.)

symmetric_difference_update(...)  
Update a set with the symmetric difference of itself and another.

union(...)  
Return the union of sets as a new set.  
(i.e. all elements that are in either set.)

update(...)  
Update a set with the union of itself and others.

**Appendix F: Dictionary Functions**

clear(...)  
D.clear() -> None. Remove all items from D.

copy(...)  
D.copy() -> a shallow copy of D
get(...)  
D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.

items(...)  
D.items() -> a set-like object providing a view on D’s items

keys(...)  
D.keys() -> a set-like object providing a view on D’s keys

pop(...)  
D.pop(k[,d]) -> v, remove specified key and return the corresponding value.  
If key is not found, d is returned if given, otherwise KeyError is raised

popitem(...)  
D.popitem() -> (k, v), remove and return some (key, value) pair as a  
2-tuple; but raise KeyError if D is empty.

setdefault(...)  
D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update(...)  
D.update(E, **F) -> None.  Update D from dict/iterable E and F.  
If E has a .keys() method, does: for k in E: D[k] = E[k]  
If E lacks .keys() method, does: for (k, v) in E: D[k] = v  
In either case, this is followed by: for k in F: D[k] = F[k]

values(...)  
D.values() -> an object providing a view on D’s values

Appendix G: File I/O

open(file, mode='r', buffering=-1, encoding=None,  
    errors=None, newline=None, closefd=True) -> file object

Open file and return a stream.  Raise IOError upon failure.

file is either a text or byte string giving the name (and the path  
if the file isn’t in the current working directory) of the file to  
be opened or an integer file descriptor of the file to be  
wrapped.  (If a file descriptor is given, it is closed when the  
returned I/O object is closed, unless closefd is set to False.)
mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

```
Character Meaning
--------- ---------------------------------------------------------------
r'     open for reading (default)
w'     open for writing, truncating the file first
'a'    open for writing, appending to the end of the file if it exists
'b'    binary mode
t'     text mode (default)
'+'    open a disk file for updating (reading and writing)
'U'    universal newline mode (for backwards compatibility; unneeded for new code)
```

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

* Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.

* "Interactive" text files (files for which isatty() returns True)
use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

events is an optional string that specifies how encoding errors are to be handled---this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, ',', '\n', '\r', and '\r\n'. It works as follows:

* On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is ',', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

* On output, if newline is None, any '\n' characters written are translated to the system default line separator, os.linesep. If newline is ',', no translation takes place. If newline is any of the other legal values, any '\n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.
It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

close()
    Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

    As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed
    True if the stream is closed.

fileno()
    Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError` is raised if the IO object does not use a file descriptor.

flush()
    Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()
    Return True if the stream is interactive (i.e., connected to a terminal/tty device).

readable()
    Return True if the stream can be read from. If False, read() will raise `OSError`.

readline(limit=-1)
    Read and return one line from the stream. If `limit` is specified, at most `limit` bytes will be read.

    The line terminator is always b\n for binary files; for text files, the newlines argument to open() can be used to select the line terminator(s) recognized.

readlines(hint=-1)
    Read and return a list of lines from the stream. `hint` can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds `hint`.

    Note that it’s already possible to iterate on file objects using for line in file: ... without calling file.readlines().

seek(offset, whence=SEEK_SET)
    Change the stream position to the given byte offset. offset is interpreted relative to the
position indicated by whence. Values for whence are:

- SEEK_SET or 0 – start of the stream (the default); offset should be zero or positive
- SEEK_CUR or 1 – current stream position; offset may be negative
- SEEK_END or 2 – end of the stream; offset is usually negative

Return the new absolute position.

*New in version 3.1:* The SEEK_* constants.

*New in version 3.3:* Some operating systems could support additional values, like os.SEEK_HOLE or os.SEEK_DATA. The valid values for a file could depend on it being open in text or binary mode.

**seekable()**

Return True if the stream supports random access. If False, seek(), tell() and truncate() will raise OSError.

**tell()**

Return the current stream position.

**truncate(size=None)**

Resize the stream to the given size in bytes (or the current position if size is not specified). The current stream position isn’t changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled, on Windows they’re undetermined). The new file size is returned.

**writable()**

Return True if the stream supports writing. If False, write() and truncate() will raise OSError.

**writelines(lines)**

Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

---

**Appendix H: Pickle Module**

**dump(...)**

```
dump(obj, file, protocol=None, *, fix_imports=True) -> None
```

Write a pickled representation of obj to the open file object file. This is equivalent to `Pickler(file, protocol).dump(obj)`", but may be more
efficient.

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

The file argument must have a write() method that accepts a single bytes argument. It can thus be a file object opened for binary writing, a io.BytesIO instance, or any other custom object that meets this interface.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python 3.x names to the old module names used in Python 2.x, so that the pickle data stream is readable with Python 2.x.

dumps(...)
dumps(obj, protocol=None, *, fix_imports=True) -> bytes

Return the pickled representation of the object as a bytes object, instead of writing it to a file.

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

If fix_imports is True and *protocol* is less than 3, pickle will try to
map the new Python 3.x names to the old module names used in Python 2.x, so that the pickle data stream is readable with Python 2.x.

load(…)
load(file, *, fix_imports=True, encoding='ASCII', errors='strict') -> object

Read a pickled object representation from the open file object file and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`", but may be more efficient.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

The argument file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus *file* can be a binary file object opened for reading, a BytesIO object, or any other custom object that meets this interface.

Optional keyword arguments are fix_imports, encoding and errors, which are used to control compatibility support for pickle stream generated by Python 2.x. If fix_imports is True, pickle will try to map the old Python 2.x names to the new names used in Python 3.x. The encoding and errors tell pickle how to decode 8-bit string instances pickled by Python 2.x; these default to 'ASCII' and 'strict', respectively.

loads(…)
loads(input, *, fix_imports=True, encoding='ASCII', errors='strict') -> object

Read a pickled object hierarchy from a bytes object and return the reconstituted object hierarchy specified therein.
The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

Optional keyword arguments are fix_imports, encoding and errors, which are used to control compatibility support for pickle stream generated by Python 2.x. If fix_imports is True, pickle will try to map the old Python 2.x names to the new names used in Python 3.x. The encoding and errors tell pickle how to decode 8-bit string instances pickled by Python 2.x; these default to 'ASCII' and 'strict', respectively.