Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*x*)

Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

aiter(*async_iterable*)

Return an asynchronous iterator for an asynchronous iterable. Equivalent to calling `x.__aiter__()`.

Note: Unlike `iter()`, `aiter()` has no 2-argument variant.

New in version 3.10.

all(*iterable*)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
```

```
    return True
```

awaitable **anext**(*async_iterator*)
awaitable **anext**(*async_iterator*, *default*)

When awaited, return the next item from the given asynchronous iterator, or *default* if given and the iterator is exhausted.

This is the async variant of the `next()` builtin, and behaves similarly.

This calls the `__anext__()` method of *async_iterator*, returning an awaitable. Awaiting this returns the next value of the iterator. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopAsyncIteration` is raised.

New in version 3.10.

any(*iterable*)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u`, or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2.

bin(*x*)

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If the prefix “0b” is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also `format()` for more information.

class **bool**(*x=False*)

Return a Boolean value, i.e. one of `True` or `False`. *x* is converted using the standard truth testing procedure. If *x* is false or omitted, this returns `False`; otherwise, it returns `True`. The `bool` class is a subclass of `int` (see Numeric Types — int, float, complex). It cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Type - bool).

breakpoint(**args, **kws*)

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import `pdb` or type as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and `breakpoint()` will automatically call that, allowing you to drop into the debugger of choice. If `sys.breakpointhook()` is not accessible, this function will raise `RuntimeError`.

By default, the behavior of `breakpoint()` can be changed with the `PYTHONBREAKPOINT` environment variable. See `sys.breakpointhook()` for usage details.

Note that this is not guaranteed if `sys.breakpointhook()` has been replaced.

Raises an auditing event `builtins.breakpoint` with argument `breakpointhook`.

New in version 3.7.

class **bytearray**(*source=b''*)
class **bytearray**(*source, encoding*)
class **bytearray**(*source, encoding, errors*)

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in Mutable Sequence Types, as well as most methods that the `bytes` type has, see Bytes and Bytearray Operations.

The optional *source* parameter can be used to initialize the array in a few different ways:

- If it is a *string*, you must also give the *encoding* (and optionally, *errors*) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an *integer*, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an *iterable*, it must be an iterable of integers in the range `0 <= x < 256`, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also Binary Sequence Types — bytes, bytearray, memoryview and Bytearray Objects.

class **bytes**(*source=b''*)
class **bytes**(*source, encoding*)
class **bytes**(*source, encoding, errors*)

Return a new "bytes" object which is an immutable sequence of integers in the range `0 <= x < 256`. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for `bytearray()`.

Bytes objects can also be created with literals, see String and Bytes literals.

See also Binary Sequence Types — bytes, bytearray, memoryview, Bytes Objects, and Bytes and Bytearray Operations.

possible that a call fails, but if it is `False`, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr(*i*)

Return the string representing a character whose Unicode code point is the integer *i*. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. This is the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if *i* is outside that range.

@classmethod

Transform a method into a class method.

A class method receives the class as an implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

The `@classmethod` form is a function decorator – see Function definitions for details.

A class method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section. For more information on class methods, see The standard type hierarchy.

Changed in version 3.9: Class methods can now wrap other descriptors such as `property()`.

Changed in version 3.10: Class methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`) and have a new `__wrapped__` attribute.

Changed in version 3.11: Class methods can no longer wrap other descriptors such as `property()`.

compile(*source, filename, mode, flags=0, dont_inherit=False, optimize=- 1*)

Compile the *source* into a code or AST object. Code objects can be executed by `exec()` or `eval()`. *source* can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file (`'<string>'` is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be `'exec'` if *source* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to

The optional arguments *flags* and *dont_inherit* control which compiler options should be activated and which future features should be allowed. If neither is present (or both are zero) the code is compiled with the same flags that affect the code that is calling `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the compiler options and the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the flags (future features and compiler options) in the surrounding code are ignored.

Compiler options and future statements are specified by bits which can be bitwise ORed together to specify multiple options. The bitfield required to specify a given future feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module. Compiler flags can be found in `ast` module, with `PyCF_` prefix.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an auditing event `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

> **Note:** When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

> **Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also, input in `'exec'` mode does not have to end in a newline anymore. Added the *optimize* parameter.

Changed in version 3.5: Previously, `TypeError` was raised when null bytes were encountered in *source*.

New in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in flags to enable support for top-level `await`, `async for`, and `async with`.

class **complex**(*real=0, imag=0*)
class **complex**(*string*)

Return a complex number with the value *real* + *imag**1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is

Note: When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in Numeric Types — int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

delattr(*object, name*)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see `setattr()`).

class **dict**(***kwarg*)
class **dict**(*mapping, **kwarg*)
class **dict**(*iterable, **kwarg*)

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and Mapping Types — dict for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

dir()
dir(*object*)

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a, b*)

Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable, start=0*)

Return an enumerate object. *iterable* must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*expression, globals=None, locals=None*)

The arguments are a string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the

`__builtins__` dictionary into *globals* before passing it to `eval()`. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, *eval()* does not have access to the nested scopes (non-locals) in the enclosing environment.

The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case, pass a code object instead of a string. If the code object has been compiled with `'exec'` as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions return the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

If the given source is a string, then leading and trailing spaces and tabs are stripped.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

exec(*object, globals=None, locals=None, /, *, closure=None*)

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section File input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at the module level, globals and locals are the same dictionary. If exec gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `exec()`.

The *closure* argument specifies a closure–a tuple of cellvars. It's only valid when the *object* is a code object containing free variables. The length of the tuple must exactly match the number of free variables referenced by the code object.

Note: The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

Note: The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

Changed in version 3.11: Added the *closure* parameter.

filter(*function, iterable*)

Construct an iterator from those elements of *iterable* for which *function* is true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if function is not `None` and `(item for item in iterable if item)` if function is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* is false.

class **float**(*x=0.0*)

Return a floating point number constructed from a number or string *x*.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the `floatvalue` production rule in the following grammar, after leading and trailing whitespace characters are removed:

```
sign        ::=  "+" | "-"
infinity    ::=  "Infinity" | "inf"
nan         ::=  "nan"
digitpart   ::=  digit (["_"] digit)*
number      ::=  [digitpart] "." digitpart | digitpart ["."]
exponent    ::=  ("e" | "E") ["+" | "-"] digitpart
floatnumber ::=  number [exponent]
floatvalue  ::=  [sign] (floatnumber | infinity | nan)
```

Here `digit` is a Unicode decimal digit (character in the Unicode general category `Nd`). Case is not significant, so, for example, “inf”, “Inf”, “INFINITY”, and “iNfINity” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Examples:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

The float type is described in Numeric Types — int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__float__()` is not defined.

format(*value*, *format_spec=''*)

Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument; however, there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

The default *format_spec* is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value's `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the *format_spec* is non-empty, or if either the *format_spec* or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises `TypeError` if *format_spec* is not an empty string.

class **frozenset**(*iterable=set()*)

Return a new `frozenset` object, optionally with elements taken from *iterable*. `frozenset` is a built-in class. See `frozenset` and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

getattr(*object*, *name*)
getattr(*object*, *name*, *default*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised. *name* need not be a Python identifier (see `setattr()`).

Note: Since private name mangling happens at compilation time, one must manually mangle a

globals()

Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

hasattr(*object*, *name*)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.)

hash(*object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

> **Note:** For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine. See `__hash__` for details.

help()
help(*request*)

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the `site` module.

Changed in version 3.4: Changes to `pydoc` and `inspect` mean that the reported signatures for callables are now more comprehensive and consistent.

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
```

See also `format()` for more information.

See also `int()` for converting a hexadecimal string to an integer using a base of 16.

> **Note:** To obtain a hexadecimal string representation for a float, use the `float.hex()` method.

id(*object*)

Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in memory.

Raises an auditing event `builtins.id` with argument `id`.

input()
input(*prompt*)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Raises an auditing event `builtins.input` with argument `prompt` before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

class **int**(*x=0*)
class **int**(*x, base=10*)

Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* defines `__int__()`, `int(x)` returns `x.__int__()`. If *x* defines `__index__()`, it returns `x.__index__()`. If *x* defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an integer in radix *base*. Optionally, the string can be preceded by `+` or `-` (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

A base-n integer string contains digits, each representing a value from 0 to n-1. The values 0–9 can be represented by any Unicode decimal digit. The values 10–35 can be represented by `a` to `z` (or `A` to `Z`). The default *base* is 10. The allowed bases are 0 and 2–36. Base-2, -8, and -16 strings can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. For base 0, the string is interpreted in a similar way to an integer literal in code, in that the actual base is 2, 8, 10, or 16 as determined by the prefix. Base 0 also disallows leading zeros: `int('010', 0)` is not legal, while

The integer type is described in Numeric Types — int, float, complex.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined.

Changed in version 3.11: The delegation to `__trunc__()` is deprecated.

Changed in version 3.11: `int` string inputs and string representations can be limited to help avoid denial of service attacks. A `ValueError` is raised when the limit is exceeded while converting a string *x* to an `int` or when converting an `int` into a string would exceed the limit. See the integer string conversion length limitation documentation.

isinstance(*object, classinfo*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect, or virtual) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples) or a Union Type of multiple types, return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised. `TypeError` may not be raised for an invalid type if an earlier check succeeds.

Changed in version 3.10: *classinfo* can be a Union Type.

issubclass(*class, classinfo*)

Return `True` if *class* is a subclass (direct, indirect, or virtual) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples) or a Union Type, in which case return `True` if *class* is a subclass of any entry in *classinfo*. In any other case, a `TypeError` exception is raised.

Changed in version 3.10: *classinfo* can be a Union Type.

iter(*object*)
iter(*object, sentinel*)

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the iterable protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its `__next__()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

See also Iterator Types.

One useful application of the second form of `iter()` is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*s*)

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

CPython implementation detail: `len` raises `OverflowError` on lengths larger than `sys.maxsize`, such as `range(2 ** 100)`.

class **list**
class **list**(*iterable*)

Rather than being a function, `list` is actually a mutable sequence type, as documented in Lists and Sequence Types — list, tuple, range.

locals()

Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks. Note that at the module level, `locals()` and `globals()` are the same dictionary.

> **Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

map(*function*, *iterable*, **iterables*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

max(*iterable*, ***, *key=None*)
max(*iterable*, ***, *default*, *key=None*)
max(*arg1*, *arg2*, **args*, *key=None*)

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be `None`.

information.

min(*iterable*, *, *key=None*)
min(*iterable*, *, *default*, *key=None*)
min(*arg1*, *arg2*, **args*, *key=None*)

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be `None`.

next(*iterator*)
next(*iterator*, *default*)

Retrieve the next item from the iterator by calling its `__next__()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

class **object**

Return a new featureless object. `object` is a base for all classes. It has methods that are common to all instances of Python classes. This function does not accept any arguments.

Note: `object` does *not* have a `__dict__`, so you can't assign arbitrary attributes to an instance of the `object` class.

oct(*x*)

Convert an integer number to an octal string prefixed with "0o". The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to an octal string either with the prefix "0o" or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
```

See also `format()` for more information.

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

Open *file* and return a corresponding file object. If the file cannot be opened, an `OSError` is raised. See Reading and Writing Files for more examples of how to use this function.

file is a path-like object giving the pathname (absolute or relative to 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), `'x'` for exclusive creation, 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: `locale.getencoding()` is called to get the current locale encoding. (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
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as `bytes` objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as `str`, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

Note: Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

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 in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another

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 (whatever `locale.getencoding()` returns), but any text encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers), though any error handling name that has been registered with `codecs.register_error()` is also valid. The standard names include:

- `'strict'` to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.
- `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as low surrogate code units ranging from U+DC80 to U+DCFF. These surrogate code units will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
- `'backslashreplace'` replaces malformed data by Python's backslashed escape sequences.
- `'namereplace'` (also only supported when writing) replaces unsupported characters with `\N{...}` escape sequences.

newline determines how to parse newline characters from the stream. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, 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 newlines 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.
- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If *newline* is `''` or `'\n'`, 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` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* must be `True` (the default); otherwise, an error will be raised.

(passing `os.open` as *opener* results in functionality similar to passing `None`).

The newly created file is non-inheritable.

The following example uses the dir_fd parameter of the `os.open()` function to open a file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

The type of file object returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a subclass of `io.TextIOBase` (specifically `io.TextIOWrapper`). When used to open a file in a binary mode with buffering, the returned class is a subclass of `io.BufferedIOBase`. The exact class varies: in read binary mode, it returns an `io.BufferedReader`; in write binary and append binary modes, it returns an `io.BufferedWriter`, and in read/write mode, it returns an `io.BufferedRandom`. When buffering is disabled, the raw stream, a subclass of `io.RawIOBase`, `io.FileIO`, is returned.

See also the file handling modules, such as `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

Raises an auditing event `open` with arguments `file`, `mode`, `flags`.

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The *opener* parameter was added.
- The `'x'` mode was added.
- `IOError` used to be raised, it is now an alias of `OSError`.
- `FileExistsError` is now raised if the file opened in exclusive creation mode (`'x'`) already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).
- The `'namereplace'` error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing `os.PathLike`.
- On Windows, opening a console buffer may return a subclass of `io.RawIOBase` other than

Changed in version 3.11: The `'U'` mode has been removed.

ord(*c*)

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of `chr()`.

pow(*base, exp, mod=None*)

Return *base* to the power *exp*; if *mod* is present, return *base* to the power *exp*, modulo *mod* (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type `int` or `float` and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`.

For `int` operands *base* and *exp*, if *mod* is present, *mod* must also be of integer type and *mod* must be nonzero. If *mod* is present and *exp* is negative, *base* must be relatively prime to *mod*. In that case, `pow(inv_base, -exp, mod)` is returned, where *inv_base* is an inverse to *base* modulo *mod*.

Here's an example of computing an inverse for `38` modulo `97`:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Changed in version 3.8: For `int` operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.8: Allow keyword arguments. Formerly, only positional arguments were supported.

print(**objects, sep=' ', end='\n', file=None, flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

fget is a function for getting an attribute value. *fset* is a function for setting an attribute value. *fdel* is a function for deleting an attribute value. And *doc* creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If *c* is an instance of *C*, `c.x` will invoke the getter, `c.x = value` will invoke the setter, and `del c.x` the deleter.

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a decorator:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for *voltage* to “Get the current voltage.”

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value
```

```
        del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

class **range**(*stop*)
class **range**(*start, stop, step=1*)

Rather than being a function, `range` is actually an immutable sequence type, as documented in Ranges and Sequence Types — list, tuple, range.

repr(*object*)

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. If `sys.displayhook()` is not accessible, this function will raise `RuntimeError`.

reversed(*seq*)

Return a reverse iterator. *seq* must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`).

round(*number, ndigits=None*)

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

For a general Python object `number`, `round` delegates to `number.__round__`.

> **Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating Point Arithmetic: Issues and Limitations for more information.

class **set**
class **set**(*iterable*)

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built-in class. See `set` and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the

setattr(*object*, *name*, *value*)

This is the counterpart of `getattr()`. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in Identifiers and keywords unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through `getattr()` etc..

> **Note:** Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with `setattr()`.

class **slice**(*stop*)
class **slice**(*start*, *stop*, *step=None*)

Return a slice object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`. Slice objects have read-only data attributes `start`, `stop`, and `step` which merely return the argument values (or their default). They have no other explicit functionality; however, they are used by NumPy and other third-party packages. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator.

Changed in version 3.12: Slice objects are now hashable (provided `start`, `stop`, and `step` are hashable).

sorted(*iterable*, */*, ***, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style *cmp* function to a *key* function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

The sort algorithm uses only `<` comparisons between items. While defining an `__lt__()` method will suffice for sorting, **PEP 8** recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as `max()` that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call reflected the `__gt__()` method.

For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

The `@staticmethod` form is a function decorator – see Function definitions for details.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). Moreover, they can be called as regular functions (such as `f()`).

Static methods in Python are similar to those found in Java or C++. Also, see `classmethod()` for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

For more information on static methods, see The standard type hierarchy.

Changed in version 3.10: Static methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`), have a new `__wrapped__` attribute, and are now callable as regular functions.

class **str**(*object=''*)
class **str**(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Return a `str` version of *object*. See `str()` for details.

`str` is the built-in string class. For general information about strings, see Text Sequence Type — str.

sum(*iterable*, */*, *start=0*)

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

Changed in version 3.8: The *start* parameter can be specified as a keyword argument.

Changed in version 3.12: Summation of floats switched to an algorithm that gives higher accuracy on most builds.

class **super**

accessing inherited methods that have been overridden in a class.

The *object_or_type* determines the method resolution order to be searched. The search starts from the class right after the *type*.

For example, if `__mro__` of *object_or_type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the *object_or_type* lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

In addition to method lookups, `super()` also works for attribute lookups. One possible use case for this is calling descriptors in a parent or sibling class.

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see guide to using super().

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in Tuples and Sequence Types — list, tuple, range.

class **type**(*object*)
class **type**(*name*, *bases*, *dict*, ***kwds*)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute; if empty, `object`, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical `type` objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

See also Type Objects.

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

See also Customizing class creation.

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()
vars(*object*)

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).

zip(**iterables*, *strict=False*)

Iterate over several iterables in parallel, producing tuples with an item from each one.

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

More formally: `zip()` returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument iterables.

Another way to think of `zip()` is that it turns rows into columns, and columns into rows. This is similar to transposing a matrix.

`zip()` is lazy: The elements won't be processed until the iterable is iterated on, e.g. by a `for` loop or by wrapping in a `list`.

One thing to consider is that the iterables passed to `zip()` could have different lengths; sometimes by design, and sometimes because of a bug in the code that prepared these iterables. Python offers three different approaches to dealing with this issue:

- By default, `zip()` stops when the shortest iterable is exhausted. It will ignore the remaining items in the longer iterables, cutting off the result to the length of the shortest iterable:

  ```
  >>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
  [(0, 'fee'), (1, 'fi'), (2, 'fo')]
  ```

- `zip()` is often used in cases where the iterables are assumed to be of equal length. In such cases, it's recommended to use the `strict=True` option. Its output is the same as regular `zip()`:

  ```
  >>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
  [('a', 1), ('b', 2), ('c', 3)]
  ```

  Unlike the default behavior, it raises a `ValueError` if one iterable is exhausted before the others:

  ```
  >>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
  ...     print(item)
  ...
  (0, 'fee')
  (1, 'fi')
  (2, 'fo')
  Traceback (most recent call last):
    ...
  ValueError: zip() argument 2 is longer than argument 1
  ```

  Without the `strict=True` argument, any bug that results in iterables of different lengths will be silenced, possibly manifesting as a hard-to-find bug in another part of the program.

- Shorter iterables can be padded with a constant value to make all the iterables have the same length. This is done by `itertools.zip_longest()`.

Edge cases: With a single iterable argument, `zip()` returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

Tips and tricks:

repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

- `zip()` in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

Changed in version 3.10: Added the `strict` argument.

__import__(*name, globals=None, locals=None, fromlist=(), level=0*)

Note: This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. `0` (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see **PEP 328** for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

Changed in version 3.3: Negative values for *level* are no longer supported (which also changes the default value to 0).

Changed in version 3.9: When the command line options `-E` or `-I` are being used, the environment variable `PYTHONCASEOK` is now ignored.

Footnotes

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

proposed by **PEP 572**.

6.2.8. Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression ::=  "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for the generator object (in the same fashion as normal generators). However, the iterable expression in the leftmost `for` clause is immediately evaluated, so that an error produced by it will be emitted at the point where the generator expression is defined, rather than at the point where the first value is retrieved. Subsequent `for` clauses and any filter condition in the leftmost `for` clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `(x*y for x in range(10) for y in range(x, x+10))`.

The parentheses can be omitted on calls with only one argument. See section Calls for details.

To avoid interfering with the expected operation of the generator expression itself, `yield` and `yield from` expressions are prohibited in the implicitly defined generator.

If a generator expression contains either `async for` clauses or `await` expressions it is called an *asynchronous generator expression*. An asynchronous generator expression returns a new asynchronous generator object, which is an asynchronous iterator (see Asynchronous Iterators).

New in version 3.6: Asynchronous generator expressions were introduced.

Changed in version 3.7: Prior to Python 3.7, asynchronous generator expressions could only appear in `async def` coroutines. Starting with 3.7, any function can use asynchronous generator expressions.

Changed in version 3.8: `yield` and `yield from` prohibited in the implicitly nested scope.

6.2.9. Yield expressions

```
yield_atom       ::=  "(" yield_expression ")"
yield_expression ::=  "yield" [expression_list | "from" expression]
```

The yield expression is used when defining a generator function or an asynchronous generator function and thus can only be used in the body of a function definition. Using a yield expression in a function's body causes that function to be a generator function, and using it in an `async def` function's body causes that coroutine function to be an asynchronous generator function. For example:

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

Due to their side effects on the containing scope, `yield` expressions are not permitted as part of the implicitly defined scopes used to implement comprehensions and generator expressions.

Changed in version 3.8: Yield expressions prohibited in the implicitly nested scopes used to implement comprehensions and generator expressions.

Generator functions are described below, while asynchronous generator functions are described separately in section Asynchronous generator functions.

proceeds to the first yield expression, where it is suspended again, returning the value of `expression_list` to the generator's caller, or `None` if `expression_list` is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If `__next__()` is used (typically via either a `for` or the `next()` builtin) then the result is `None`. Otherwise, if `send()` is used, then the result will be the value passed in to that method.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where the execution should continue after it yields; the control is always transferred to the generator's caller.

Yield expressions are allowed anywhere in a `try` construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending `finally` clauses to execute.

When `yield from <expr>` is used, the supplied expression must be an iterable. The values produced by iterating that iterable are passed directly to the caller of the current generator's methods. Any values passed in with `send()` and any exceptions passed in with `throw()` are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then `send()` will raise `AttributeError` or `TypeError`, while `throw()` will just raise the passed in exception immediately.

When the underlying iterator is complete, the `value` attribute of the raised `StopIteration` instance becomes the value of the yield expression. It can be either set explicitly when raising `StopIteration`, or automatically when the subiterator is a generator (by returning a value from the subgenerator).

Changed in version 3.3: Added `yield from <expr>` to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

See also:

PEP 255 - Simple Generators
The proposal for adding generators and the `yield` statement to Python.

PEP 342 - Coroutines via Enhanced Generators
The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

PEP 380 - Syntax for Delegating to a Subgenerator
The proposal to introduce the `yield_from` syntax, making delegation to subgenerators easy.

PEP 525 - Asynchronous Generators
The proposal that expanded on **PEP 492** by adding generator capabilities to coroutine functions.

6.2.9.1. Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the generator is already executing raises a `ValueError` exception.

`generator.__next__()`
Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a `__next__()` method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**`(`*`x`*`)`
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**`(`*`iterable`*`)`
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**`(`*`i, x`*`)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**`(`*`x`*`)`
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**`([`*`i`*`])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**`()`
Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**`(`*`x`*`[, `*`start`*`[, `*`end`*`]])`
Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

`list.`**`count`**`(`*`x`*`)`
Return the number of times *x* appears in the list.

`list.`**`sort`**`(`*`*, key=None, reverse=False`*`)`
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

`list.`**`reverse`**`()`
Reverse the elements of the list in place.

`list.`**`copy`**`()`
Return a shallow copy of the list. Equivalent to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
```

```
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and *None* can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
```

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the `for` and `if` statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
```

```
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

The following list comprehension will transpose rows and columns:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the inner list comprehension is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See Unpacking Argument Lists for details on the asterisk in this line.

5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.

5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

section) or indexing (or even by attribute in the case of `namedtuples`). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions, set comprehensions are also supported:

```
{'r', 'd'}
```

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. Looping Techniques

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Using `set()` on a sequence eliminates duplicate elements. The use of `sorted()` in combination with `set()` over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
```

```
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` are membership tests that determine whether a value is in (or not in) a container. The operators `is` and `is not` compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the walrus operator `:=`. This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

5.8. Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
```

```
(1, 2)                       < (1, 2, -1)
(1, 2, 3)                   == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))         < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.

Footnotes

[1] Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.

See also

`Figure.subfigures`

`add_subplot`(**args, **kwargs*) [source]

Add an `Axes` to the figure as part of a subplot arrangement.

Call signatures:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

Parameters:

args : *int, (int, int, index), or `SubplotSpec`, default: (1, 1, 1)

The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A `SubplotSpec`.

In rare circumstances, `add_subplot` may be called with a single argument, a subplot Axes instance already created in the present figure but not in the figure's list of Axes.

projection : *{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional*

The projection type of the subplot (`Axes`). *str* is the name of a custom projection, see `projections`. The default None results in a 'rectilinear' projection.

polar : *bool, default: False*

If True, equivalent to projection='polar'.

axes_class : *subclass type of `Axes`, optional*

The `axes.Axes` subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See axisartist for examples.

sharex, sharey : *`Axes`, optional*

Share the x or y `axis` with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared axes.

label : *str*

A label for the returned Axes.

Returns:

`Axes`

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as `projections.polar.PolarAxes` for polar projections.

Other Parameters:

****kwargs**

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class `Axes` can be found in the following table but there might also be other keyword arguments if another projection is used.

Property	Description
`adjustable`	{'box', 'datalim'}
`agg_filter`	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`	scalar or None
`anchor`	(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}
`animated`	bool
`aspect`	{'auto', 'equal'} or float
`autoscale_on`	bool
`autoscalex_on`	unknown
`autoscaley_on`	unknown
`axes_locator`	Callable[[Axes, Renderer], Bbox]
`axisbelow`	bool or 'line'
`box_aspect`	float or None
`clip_box`	`BboxBase` or None
`clip_on`	bool
`clip_path`	Patch or (Path, Transform) or None
`facecolor` or fc	color
`figure`	`Figure`
`frame_on`	bool
`gid`	str
`in_layout`	bool

`in_layout`	bool
`label`	object
`mouseover`	bool
`navigate`	bool
`navigate_mode`	unknown
`path_effects`	list of `AbstractPathEffect`
`picker`	None or bool or float or callable
`position`	[left, bottom, width, height] or `Bbox`
`prop_cycle`	`Cycler`
`rasterization_zorder`	float or None
`rasterized`	bool
`sketch_params`	(scale: float, length: float, randomness: float)
`snap`	bool or None
`subplotspec`	unknown
`title`	str
`transform`	`Transform`
`url`	str
`visible`	bool
`xbound`	(lower: float, upper: float)
`xlabel`	str
`xlim`	(left: float, right: float)
`xmargin`	float greater than -0.5
`xscale`	unknown
`xticklabels`	unknown
`xticks`	unknown
`ybound`	(lower: float, upper: float)
`ylabel`	str
`ylim`	(bottom: float, top: float)
`ymargin`	float greater than -0.5
`yscale`	unknown
`yticklabels`	unknown
`yticks`	unknown
`zorder`	float

`zorder`	float

See also

- `Figure.add_axes`
- `pyplot.subplot`
- `pyplot.axes`
- `Figure.subplots`
- `pyplot.subplots`

Examples

```
fig = plt.figure()

fig.add_subplot(231)
ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general

fig.add_subplot(232, frameon=False)  # subplot with no frame
fig.add_subplot(233, projection='polar')  # polar subplot
fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with ax1
fig.add_subplot(235, facecolor="red")  # red subplot

ax1.remove()  # delete ax1 from the figure
fig.add_subplot(ax1)  # add ax1 back to the figure
```

`align_labels(axs=None)` [source]

Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

Parameters:

axs : *list of* `Axes`

Optional list (or `ndarray`) of `Axes` to align the labels. Default is to align all Axes on the figure.

See also

- `matplotlib.figure.Figure.align_xlabels`
- `matplotlib.figure.Figure.align_ylabels`

`align_xlabels(axs=None)` [source]

Align the xlabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

mpl_toolkits.mplot3d.axes3d.Axes3D.plot_tr

Axes3D.plot_trisurf(**args*, *color=None*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*) [source]

Plot a triangulated surface.

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for an explanation of these possibilities.

The remaining arguments are:

```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

Parameters:

X, Y, Z : *array-like*

Data values as 1D arrays.

color

Color of the surface patches.

cmap

A colormap for the surface patches.

norm : *Normalize*

An instance of Normalize to map values to colors.

vmin, vmax : *float, default: None*

Minimum and maximum value to map.

shade : *bool, default: True*

Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

lightsource : `LightSource`

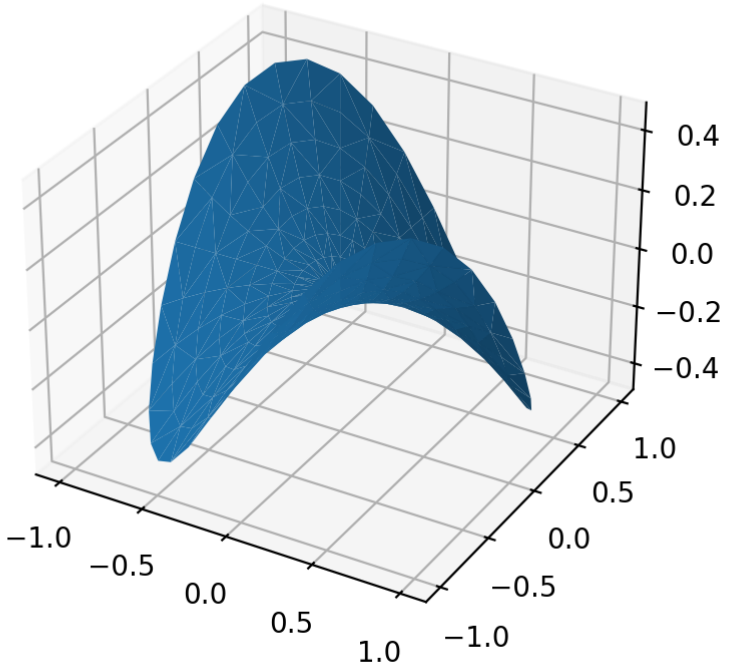
The lightsource to use when *shade* is True.

****kwargs**

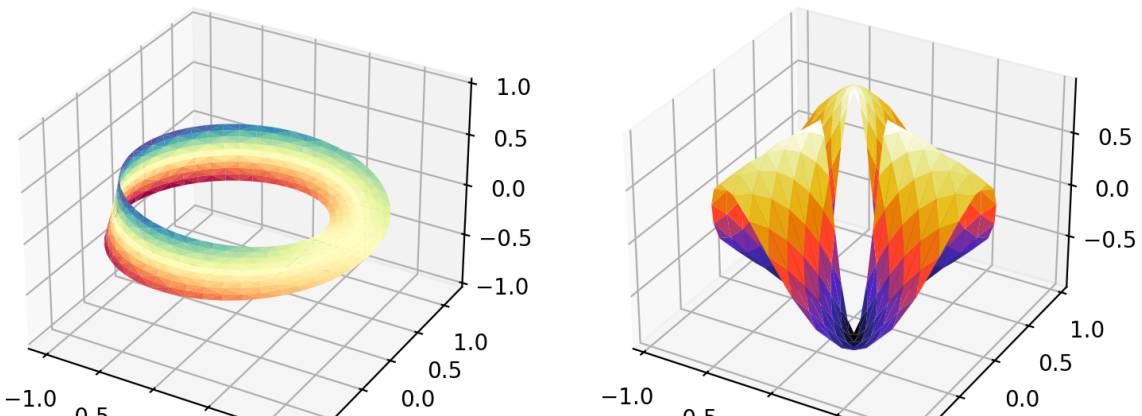
All other keyword arguments are passed on to `Poly3DCollection`

Examples

(Source code , 2x.png , png)



(Source code , 2x.png , png)



matplotlib.pyplot.clf

matplotlib.pyplot.clf() [source]

Clear the current figure.

matplotlib.pyplot.contourf

`matplotlib.pyplot.contourf(*args, data=None, **kwargs)` [source]

Plot filled contours.

Call signature:

```
contourf([X, Y,] Z, [levels], **kwargs)
```

`contour` and `contourf` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters:

X, Y : *array-like, optional*

The coordinates of the values in *Z*.

X and *Y* must both be 2D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == N` is the number of columns in *Z* and `len(Y) == M` is the number of rows in *Z*.

X and *Y* must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. `X = range(N)`, `Y = range(M)`.

Z : *(M, N) array-like*

The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

levels : *int or array-like, optional*

Determines the number and positions of the contour lines / regions.

If an int *n*, use `MaxNLocator`, which tries to automatically choose no more than *n+1* "nice" contour levels between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns:

`QuadContourSet`

Other Parameters:

corner_mask : *bool, default:* `rcParams["contour.corner_mask"]` *(default:* `True`*)*

Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

colors : *color string or sequence of colors, optional*

The colors of the levels, i.e. the lines for `contour` and the areas for `contourf`.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha : *float, default: 1*

The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap : *str or `Colormap`, default: `rcParams["image.cmap"]` (default: `'viridis'`)*

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

norm : *str or `Normalize`, optional*

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `Normalize` or one of its subclasses (see Colormap Normalization).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

vmin, vmax : *float, optional*

When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

origin : *{None, 'upper', 'lower', 'image'}, default: None*

Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`.

This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent : *(x0, x1, y0, y1), optional*

If *origin* is not *None*, then *extent* is interpreted as in `imshow`: it gives the outer pixel

If *origin* is not *None*, then *extent* is interpreted as in `imshow`. It gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].
This argument is ignored if *X* and *Y* are specified in the call to contour.

locator : *ticker.Locator subclass, optional*

The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to `MaxNLocator`.

extend : *{'neither', 'both', 'min', 'max'}, default: 'neither'*

Determines the `contourf`-coloring of values that are outside the *levels* range.
If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.
Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the `Colormap`. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `Colormap.set_under` and `Colormap.set_over`.

> **Note**
>
> An existing `QuadContourSet` does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `QuadContourSet` because it internally calls `QuadContourSet.changed()`.

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits : *registered units, optional*

Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

antialiased : *bool, optional*

Enable antialiasing, overriding the defaults. For filled contours, the default is *False*. For line contours, it is taken from `rcParams["lines.antialiased"]` (default: `True`).

nchunk : *int >= 0, optional*

If 0, no subdivision of the domain. Specify a positive integer to divide the domain into

subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

linewidths : *float or array-like, default:* `rcParams["contour.linewidth"]` ***(default:*** `None`***)***

Only applies to `contour`.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles : *{None, 'solid', 'dashed', 'dashdot', 'dotted'}, optional*

Only applies to `contour`.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the *negative_linestyles* argument.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

negative_linestyles : *{None, 'solid', 'dashed', 'dashdot', 'dotted'}, optional*

Only applies to `contour`.

If *linestyles* is *None* and the lines are monochrome, this argument specifies the line style for negative contours.

If *negative_linestyles* is *None*, the default is taken from `rcParams["contour.negative_linestyles"]`.

negative_linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches : *list[str], optional*

Only applies to `contourf`.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

algorithm : *{'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional*

Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ContourPy, consult the ContourPy documentation for further information.

The default is taken from `rcParams["contour.algorithm"]` (default: `'mpl2014'`).

clip_path : `Patch` *or* `Path` *or* `TransformedPath`

Set the clip path. See `set_clip_path`.

> ***New in version 3.8***

data : *indexable object, optional*

If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Notes

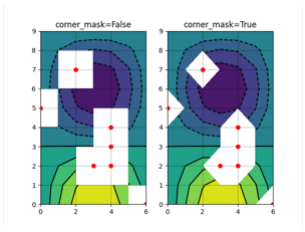
1. `contourf` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour`.
2. `contourf` fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

   ```
   z1 < Z <= z2
   ```

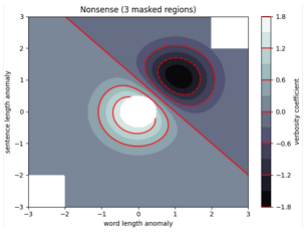
   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. `contour` and `contourf` use a marching squares algorithm to compute contour locations. More information can be found in ContourPy documentation.

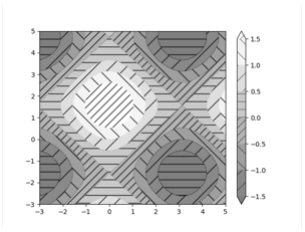
Examples using `matplotlib.pyplot.contourf`



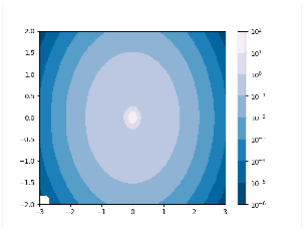
Contour Corner Mask



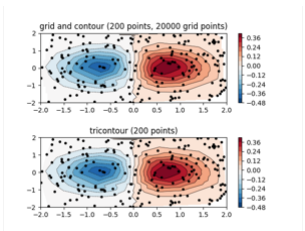
Contourf demo



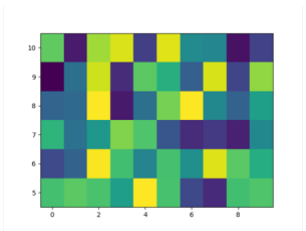
Contourf Hatching



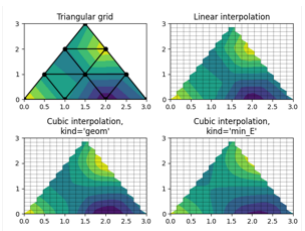
Contourf and log color scale



Contour plot of irregularly spaced data



pcolormesh



Triinterp Demo

matplotlib.pyplot.pause

matplotlib.pyplot.pause(*interval*) [source]

Run the GUI event loop for *interval* seconds.

If there is an active figure, it will be updated and displayed before the pause, and the GUI event loop (if any) will run during the pause.

This can be used for crude animation. For more complex animation use `matplotlib.animation`.

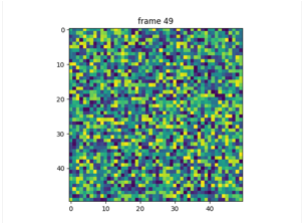
If there is no active figure, sleep for *interval* seconds instead.

See also

`matplotlib.animation`
Proper animations

`show`
Show all figures and optional block until all figures are closed.

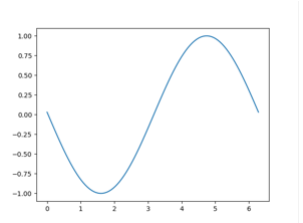
Examples using `matplotlib.pyplot.pause`



pyplot animation

Rotating a 3D plot

Animate a 3D wireframe plot



Faster rendering by using blitting

Created using Sphinx 7.2.6.

Built from v3.8.0-3-g3411279f04.

Built with the PyData Sphinx Theme 0.13.3.

matplotlib.pyplot.plot

matplotlib.pyplot.plot(args*, *scalex=True*, *scaley=True*, *data=None*, ***kwargs*)** [source]

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

You can use `Line2D` properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

When conflicting with *fmt*, keyword arguments take precedence.

Plotting labelled data

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in *x* and *y*, you can provide the object in the *data* parameter and just give the labels for *x* and *y*:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

Plotting multiple sets of data

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times. Example:

```
>>> plot(x1, y1, 'bo')
>>> plot(x2, y2, 'go')
```

- If *x* and/or *y* are 2D arrays a separate data set will be drawn for every column. If both *x* and *y* are 2D, they must have the same shape. If only one of them is 2D with shape (N, m) the other must have length N and will be used for every data set m.

  Example:

```
>>> x = [1, 2, 3]
>>> y = np.array([[1, 2], [3, 4], [5, 6]])
>>> plot(x, y)
```

  is equivalent to:

```
>>> for col in range(y.shape[1]):
...     plot(x, y[:, col])
```

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]* groups:

```
>>> plot(x1, y1, 'g^', x2, y2, 'g-')
```

  In this case, any additional keyword argument applies to all datasets. Also, this syntax cannot be combined with the *data* parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *fmt* and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

Parameters:

x, y : *array-like or scalar*

The horizontal / vertical coordinates of the data points. *x* values are optional and default to `range(len(y))`.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

fmt : *str, optional*

A format string, e.g. 'ro' for red circles. See the *Notes* section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

data : *indexable object, optional*

An object with labelled data. If given, provide the label names to plot in *x* and *y*.

> **Note**
>
> Technically there's a slight ambiguity in calls where the second label is a valid *fmt*. `plot('n', 'o', data=obj)` could be `plt(x, y)` or `plt(y, fmt)`. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string `plot('n', 'o', '', data=obj)`.

Returns:

list of `Line2D`

A list of lines representing the plotted data.

Other Parameters:

scalex, scaley : *bool, default: True*

These parameters determine if the view limits are adapted to the data limits. The values are passed on to `autoscale_view`.

****kwargs : `Line2D` *properties, optional***

kwargs are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color. Example:

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you specify multiple lines with one plot call, the kwargs apply to all those lines. In case the label object is iterable, each element is used as labels for each set of data.

Here is a list of available `Line2D` properties:

Property	Description
`agg_filter`	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`	scalar or None
`animated`	bool
`antialiased` or aa	bool
`clip_box`	`BboxBase` or None
`clip_on`	bool
`clip_path`	Patch or (Path, Transform) or None
`color` or c	color
`dash_capstyle`	`CapStyle` or {'butt', 'projecting', 'round'}
`dash_joinstyle`	`JoinStyle` or {'miter', 'round', 'bevel'}
`dashes`	sequence of floats (on/off ink in points) or (None, None)
`data`	(2, N) array or two 1D arrays
`drawstyle` or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
`figure`	`Figure`
`fillstyle`	{'full', 'left', 'right', 'bottom', 'top', 'none'}
`gapcolor`	color or None
`gid`	str
`in_layout`	bool
`label`	object
`linestyle` or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
`linewidth` or lw	float

`linewidth` or lw	float
`marker`	marker style string, `Path` or `MarkerStyle`
`markeredgecolor` or mec	color
`markeredgewidth` or mew	float
`markerfacecolor` or mfc	color
`markerfacecoloralt` or mfcalt	color
`markersize` or ms	float
`markevery`	None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
`mouseover`	bool
`path_effects`	list of `AbstractPathEffect`
`picker`	float or callable[[Artist, Event], tuple[bool, dict]]
`pickradius`	float
`rasterized`	bool
`sketch_params`	(scale: float, length: float, randomness: float)
`snap`	bool or None
`solid_capstyle`	`CapStyle` or {'butt', 'projecting', 'round'}
`solid_joinstyle`	`JoinStyle` or {'miter', 'round', 'bevel'}
`transform`	unknown
`url`	str
`visible`	bool
`xdata`	1D array
`ydata`	1D array
`zorder`	float

See also

`scatter`

> XY scatter plot with markers of varying size and/or color (sometimes also called bubble chart).

Notes

Format Strings

A format string consists of a part for color, marker and line:

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If `line` is given, but no `marker`, the data will be a line without markers.

Other combinations such as `[color][marker][line]` are also supported, but note that their parsing may be ambiguous.

Markers

character	description
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'8'`	octagon marker

`'s'`	square marker
`'p'`	pentagon marker
`'P'`	plus (filled) marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'X'`	x (filled) marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

Line Styles

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style

Example format strings:

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
'^k:'  # black triangle_up markers connected by a dotted line
```

Colors

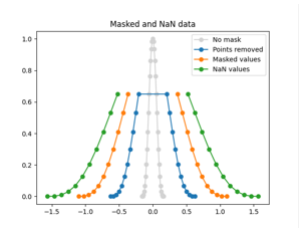
The supported color abbreviations are the single letter codes

character	color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta
`'y'`	yellow
`'k'`	black
`'w'`	white

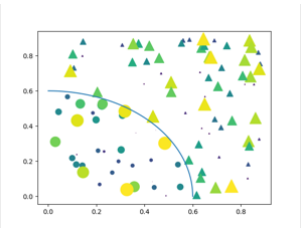
and the `'CN'` colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any `matplotlib.colors` spec, e.g. full names (`'green'`) or hex strings (`'#008000'`).

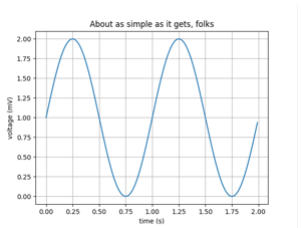
Examples using `matplotlib.pyplot.plot`



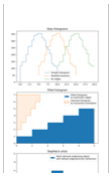
Plotting masked and NaN values

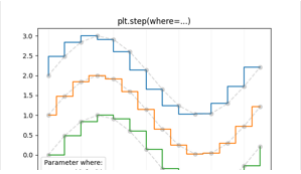


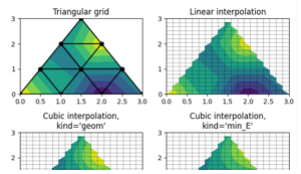
Scatter Masked



Simple Plot







matplotlib.pyplot.scatter

matplotlib.pyplot.**scatter**(*x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, *, edgecolors=None, plotnonfinite=False, data=None, **kwargs*) [source]

A scatter plot of *y* vs. *x* with varying marker size and/or color.

Parameters:

x, y : *float or array-like, shape (n,)*

The data positions.

s : *float or array-like, shape (n,), optional*

The marker size in points**2 (typographic points are 1/72 in.). Default is `rcParams['lines.markersize'] ** 2`.

The linewidth and edgecolor can visually interact with the marker size, and can lead to artifacts if the marker size is smaller than the linewidth.

If the linewidth is greater than 0 and the edgecolor is anything but *'none'*, then the effective size of the marker will be increased by half the linewidth because the stroke will be centered on the edge of the shape.

To eliminate the marker edge either set *linewidth=0* or *edgecolor='none'*.

c : *array-like or list of colors or color, optional*

The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2D array with a single row. Otherwise, value-matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the `Axes`' current "shape and fill" color cycle. This cycle defaults to `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

marker : `MarkerStyle`, *default:* `rcParams["scatter.marker"]` *(default:* `'o'`*)*

The marker style. *marker* can be either an instance of the class or the text shorthand for a

The marker style. *marker* can be either an instance of the class or the text shorthand for a particular marker. See `matplotlib.markers` for more information about marker styles.

cmap : *str or `Colormap`, default: `rcParams["image.cmap"]` (default: `'viridis'`)*

The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored if *c* is RGB(A).

norm : *str or `Normalize`, optional*

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `Normalize` or one of its subclasses (see Colormap Normalization).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if *c* is RGB(A).

vmin, vmax : *float, optional*

When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *c* is RGB(A).

alpha : *float, default: None*

The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths : *float or array-like, default: `rcParams["lines.linewidth"]` (default: `1.5`)*

The linewidth of the marker edges. Note: The default *edgecolors* is 'face'. You may want to change this as well.

edgecolors : *{'face', 'none', None} or color or sequence of color, default: `rcParams["scatter.edgecolors"]` (default: `'face'`)*

The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, *edgecolors* is ignored. Instead, the color is determined like with 'face', i.e. from *c*, *colors*, or *facecolors*.

plotnonfinite : *bool, default: False*

Whether to plot points with nonfinite *c* (i.e. `inf`, `-inf` or `nan`). If `True` the points are drawn with the *bad* colormap color (see `Colormap.set_bad`).

Returns:

`PathCollection`

Other Parameters:

data : *indexable object, optional*

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

x, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*

****kwargs** : `Collection` *properties*

See also

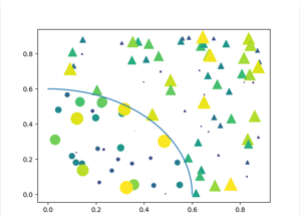
`plot`

To plot scatter plots when markers are identical in size and color.

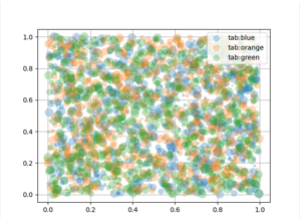
Notes

- The `plot` function will be faster for scatterplots where markers don't vary in size or color.
- Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.
- Fundamentally, scatter works with 1D arrays; *x*, *y*, *s*, and *c* may be input as N-D arrays, but within scatter they will be flattened. The exception is *c*, which will be flattened only if its size matches the size of *x* and *y*.

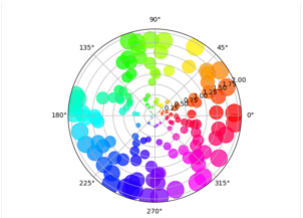
Examples using `matplotlib.pyplot.scatter`



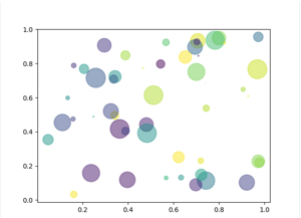
Scatter Masked



Scatter plots with a legend

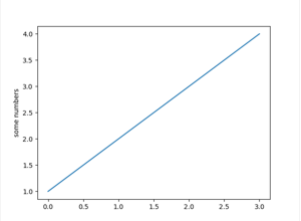


Scatter plot on polar axis



Scatter plot

Hyperlinks



Pyplot tutorial

matplotlib.pyplot.show

`matplotlib.pyplot.show(*, block=None)` [source]

Display all open figures.

Parameters:

block : *bool, optional*

Whether to wait for all figures to be closed before returning.

If `True` block and run the GUI main loop until all figure windows are closed.

If `False` ensure that all figure windows are displayed and return immediately. In this case, you are responsible for ensuring that the event loop is running to have responsive figures.

Defaults to True in non-interactive mode and to False in interactive mode (see `pyplot.isinteractive`).

See also

`ion`
: Enable interactive mode, which shows / updates the figure after every plotting command, so that calling `show()` is not necessary.

`ioff`
: Disable interactive mode.

`savefig`
: Save the figure to an image file instead of showing it on screen.

Notes

Saving figures to file and showing a window at the same time

If you want an image file as well as a user interface window, use `pyplot.savefig` before `pyplot.show`. At the end of (a blocking) `show()` the figure is closed and thus unregistered from pyplot. Calling `pyplot.savefig` afterwards would save a new and thus empty figure. This limitation of command order does not apply if the show is non-blocking or if you keep a reference to the figure and use `Figure.savefig`.

Auto-show in jupyter notebooks

The jupyter backends (activated via `%matplotlib inline`, `%matplotlib notebook`, or `%matplotlib widget`), call `show()` at the end of every cell by default. Thus, you usually don't have to call it explicitly there.

matplotlib.pyplot.title

matplotlib.pyplot.title(*label, fontdict=None, loc=None, pad=None, *, y=None, **kwargs*) [source]

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

Parameters:

label : *str*

Text to use for the title

fontdict : *dict*

> **Discouraged**
>
> The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_title(..., **fontdict)`.

A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc : *{'center', 'left', 'right'}, default: `rcParams["axes.titlelocation"]` (default: `'center'`)*

Which title to set.

y : *float, default: `rcParams["axes.titley"]` (default: `None`)*

Vertical Axes location for the title (1.0 is the top). If None (the default) and `rcParams["axes.titley"]` (default: `None`) is also None, y is determined automatically to avoid decorators on the Axes.

pad : *float, default: `rcParams["axes.titlepad"]` (default: `6.0`)*

The offset of the title from the top of the Axes, in points.

Returns:

`Text`

The matplotlib.text.Text instance representing the title

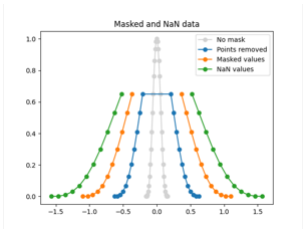
The matplotlib text instance representing the title

Other Parameters:

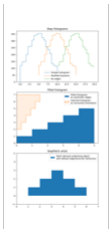
****kwargs : `Text` *properties***

Other keyword arguments are text properties, see `Text` for a list of valid text properties.

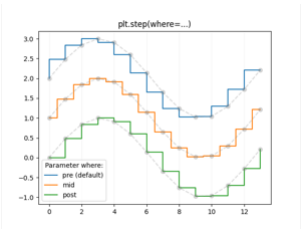
Examples using `matplotlib.pyplot.title`



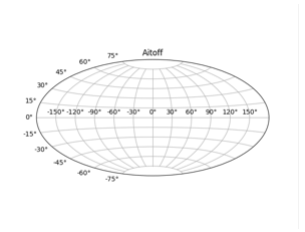
Plotting masked and NaN values



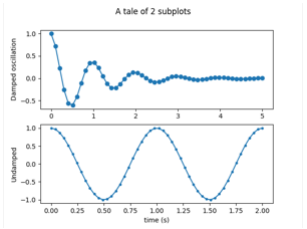
Stairs Demo



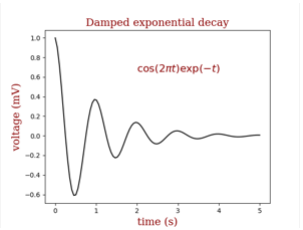
Step Demo



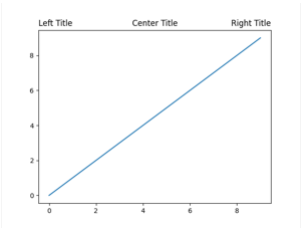
Geographic Projections



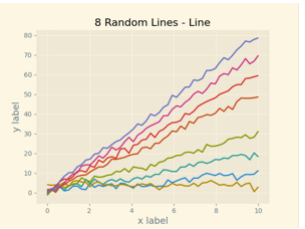
Multiple subplots



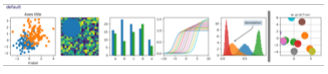
Controlling style of text and labels using a dictionary



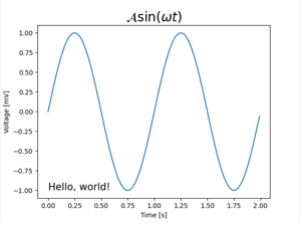
Title positioning



Solarized Light stylesheet

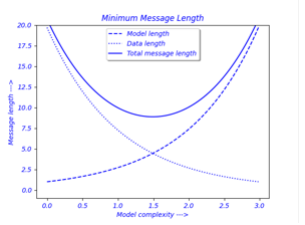


Style sheets reference



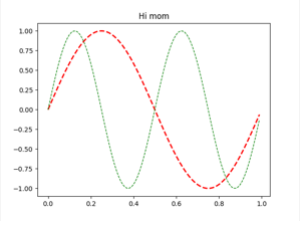
Text and mathtext using pyplot

Interactive functions



Findobj Demo

Multipage PDF



Set and get properties

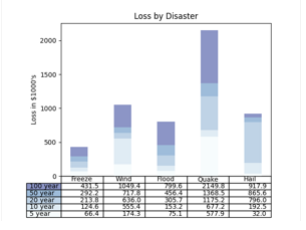
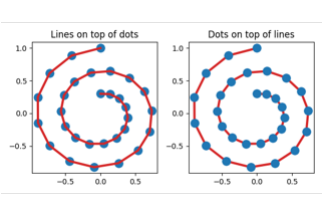


Table Demo



Zorder Demo

matplotlib.pyplot.ylim

`matplotlib.pyplot.ylim(*args, **kwargs)` [source]

Get or set the y-limits of the current axes.

Call signatures:

```
bottom, top = ylim()  # return the current ylim
ylim((bottom, top))   # set the ylim to bottom, top
ylim(bottom, top)     # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass *bottom* or *top* as kwargs, i.e.:

```
ylim(top=3)  # adjust the top leaving bottom unchanged
ylim(bottom=1)  # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

Returns:

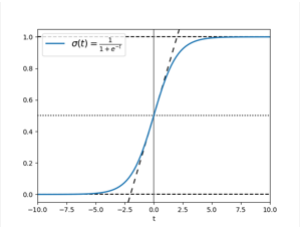
bottom, top

A tuple of the new y-axis limits.

Notes

Calling this function with no arguments (e.g. `ylim()`) is the pyplot equivalent of calling `get_ylim` on the current axes. Calling this function with arguments is the pyplot equivalent of calling `set_ylim` on the current axes. All arguments are passed though.

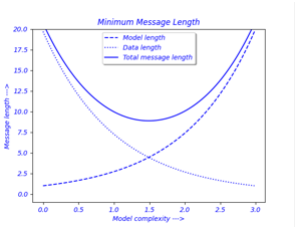
Examples using `matplotlib.pyplot.ylim`



Infinite lines

Frame grabbing

Interactive functions



Findobj Demo

numpy.arange

numpy.arange([*start*,]*stop*, [*step*,]*dtype=None*, *, *like=None*)

Return evenly spaced values within a given interval.

`arange` can be called with a varying number of positional arguments:

- `arange(stop)`: Values are generated within the half-open interval `[0, stop)` (in other words, the interval including *start* but excluding *stop*).
- `arange(start, stop)`: Values are generated within the half-open interval `[start, stop)`.
- `arange(start, stop, step)` Values are generated within the half-open interval `[start, stop)`, with spacing between values given by `step`.

For integer arguments the function is roughly equivalent to the Python built-in **`range`**, but returns an ndarray rather than a `range` instance.

When using a non-integer step, such as 0.1, it is often better to use **`numpy.linspace`**.

See the Warning sections below for more information.

Parameters:

start : *integer or real, optional*

Start of interval. The interval includes this value. The default start value is 0.

stop : *integer or real*

End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step : *integer or real, optional*

Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

dtype : *dtype, optional*

The type of the output array. If **`dtype`** is not given, infer the data type from the other input arguments.

like : *array_like, optional*

Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

New in version 1.20.0.

Returns:

arange : *ndarray*

Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

Warning

The length of the output might not be numerically stable.

Another stability issue is due to the internal implementation of `numpy.arange`. The actual step value used to populate the array is `dtype(start + step) - dtype(start)` and not *step*. Precision loss can occur here, due to casting or due to using floating points when *start* is much larger than *step*. This can lead to unexpected behaviour. For example:

```
>>> np.arange(0, 5, 0.5, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.arange(-3, 3, 0.5, dtype=int)
array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

In such cases, the use of `numpy.linspace` should be preferred.

The built-in `range` generates Python built-in integers that have arbitrary size, while `numpy.arange` produces `numpy.int32` or `numpy.int64` numbers. This may result in incorrect results for large integer values:

```
>>> power = 40
>>> modulo = 10000
>>> x1 = [(n ** power) % modulo for n in range(8)]
>>> x2 = [(n ** power) % modulo for n in np.arange(8)]
>>> print(x1)
[0, 1, 7776, 8801, 6176, 625, 6576, 4001]  # correct
>>> print(x2)
[0, 1, 7776, 7185, 0, 5969, 4816, 3361]  # incorrect
```

See also

`numpy.linspace`
: Evenly spaced numbers with careful handling of endpoints.

`numpy.ogrid`
: Arrays of evenly spaced numbers in N-dimensions.

`numpy.mgrid`
: Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Previous: How to create arrays with regularly-spaced values
numpy.core.defchararray.asarray

Next
numpy.linspace

Examples

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

The N-dimensional array (`ndarray`)

An **`ndarray`** is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its **`shape`**, which is a **`tuple`** of *N* non-negative integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype), one of which is associated with each ndarray.

As with other container objects in Python, the contents of an **`ndarray`** can be accessed and modified by indexing or slicing the array (using, for example, *N* integers), and via the methods and attributes of the **`ndarray`**.

Different **`ndarrays`** can share the same data, so that changes made in one **`ndarray`** may be visible in another. That is, an ndarray can be a *"view"* to another ndarray, and the data it is referring to is taken care of by the *"base"* ndarray. ndarrays can also be views to memory owned by Python **`strings`** or objects implementing the **`buffer`** or array interfaces.

Example

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<class 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

The array can be indexed using Python container-like syntax:

```
>>> # The element of x in the *second* row, *third* column, namely, 6.
>>> x[1, 2]
6
```

For example slicing can produce views of the array:

```
>>> y = x[:,1]
>>> y
array([2, 5], dtype=int32)
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5], dtype=int32)
>>> x
array([[1, 9, 3],
       [4, 5, 6]], dtype=int32)
```

Constructing arrays

New arrays can be constructed using the routines detailed in Array creation routines, and also by using the low-level **`ndarray`** constructor:

`ndarray`(shape[, dtype, buffer, offset, ...])	An array object represents a multidimensional, homogeneous array of fixed-size items.

Indexing arrays

Arrays can be indexed using an extended Python slicing syntax, `array[selection]`. Similar syntax is also used for accessing fields in a structured data type.

> **See also**
>
> Array Indexing.

Internal memory layout of an ndarray

An instance of class **`ndarray`** consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps *N* integers into the location of an item in the block. The ranges in which the indices can vary is specified by the **`shape`** of the array. How many bytes each item takes and how the bytes are interpreted is defined by the data-type object associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an *N*-dimensional array in a 1-dimensional block. NumPy is flexible, and **`ndarray`** objects can accommodate any *strided indexing scheme*. In a strided scheme, the N-dimensional index $(n_0, n_1, \ldots, n_{N-1})$ corresponds to the offset (in bytes):

$$n_{\text{offset}} = \sum_{k=0}^{N-1} s_k n_k$$

from the beginning of the memory block associated with the array. Here, s_k are integers which specify the **`strides`** of the array. The column-major order (used, for example, in the Fortran language and in *Matlab*) and row-major order (used in C) schemes are just specific kinds of strided scheme, and correspond to memory that can be *addressed* by the strides:

$$s_k^{\text{column}} = \text{itemsize} \prod_{j=0}^{k-1} d_j, \quad s_k^{\text{row}} = \text{itemsize} \prod_{j=k+1}^{N-1} d_j .$$

where d_j = *self.shape[j]*.

Both the C and Fortran orders are contiguous, *i.e.,* single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

> **Note**
>
> *Contiguous arrays* and *single-segment arrays* are synonymous and are used interchangeably throughout the documentation.

While a C-style and Fortran-style contiguous array, which has the corresponding flags set, can be addressed with the above strides, the actual strides may be different. This can happen in two cases:

1. If `self.shape[k] == 1` then for any legal index `index[k] == 0`. This means that in the formula for the offset $n_k = 0$ and thus $s_k n_k = 0$ and the value of s_k = *self.strides[k]* is arbitrary.
2. If an array has no elements (`self.size == 0`) there is no legal index and the strides are never used. Any array with no elements may be considered C-style and Fortran-style contiguous.

Point 1. means that `self` and `self.squeeze()` always have the same contiguity and `aligned` flags value. This also means that even a high dimensional array could be C-style and Fortran-style contiguous at the same time.

An array is considered aligned if the memory offsets for all elements and the base offset itself is a multiple of *self.itemsize*. Understanding *memory-alignment* leads to better performance on most hardware.

> **Warning**
>
> It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.
>
> `NPY_RELAXED_STRIDES_DEBUG=1` can be used to help find errors when incorrectly relying on the strides in C-extension code (see below warning).

Data in new **`ndarrays`** is in the row-major (C) order, unless otherwise specified, but, for example, basic array slicing often produces views in a different scheme.

> **Note**
>
> Several algorithms in NumPy work on arbitrarily strided arrays. However, some algorithms require single-segment arrays. When an irregularly strided array is passed in to such algorithms, a copy is automatically made.

Array attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Information on each attribute is given below.

Memory layout

The following attributes contain information about the memory layout of the array:

`ndarray.flags`	Information about the memory layout of the array.
`ndarray.shape`	Tuple of array dimensions.
`ndarray.strides`	Tuple of bytes to step in each dimension when traversing an array.
`ndarray.ndim`	Number of array dimensions.
`ndarray.data`	Python buffer object pointing to the start of the array's data.
`ndarray.size`	Number of elements in the array.
`ndarray.itemsize`	Length of one array element in bytes.
`ndarray.nbytes`	Total bytes consumed by the elements of the array.
`ndarray.base`	Base object if memory is from some other object.

Data type

> **See also**
>
> Data type objects

The data type object associated with the array can be found in the `dtype` attribute:

`ndarray.dtype`	Data-type of the array's elements.

Other attributes

`ndarray.T`	View of the transposed array.
`ndarray.real`	The real part of the array.
`ndarray.imag`	The imaginary part of the array.
`ndarray.flat`	A 1-D iterator over the array.

Array interface

> **See also**
>
> The array interface protocol.

`__array_interface__`	Python-side of the array interface
`__array_struct__`	C-side of the array interface

`ctypes` foreign function interface

`ndarray.ctypes`	An object to simplify the interaction of the array with the ctypes module.

Array methods

An `ndarray` object has many methods which operate on or with the array in some fashion, typically returning an array result. These methods are briefly explained below. (Each method's docstring has a more complete description.)

For the following methods there are also corresponding functions in `numpy`: `all`, `any`, `argmax`, `argmin`, `argpartition`, `argsort`, `choose`, `clip`, `compress`, `copy`, `cumprod`, `cumsum`, `diagonal`, `imag`, `max`, `mean`, `min`, `nonzero`, `partition`, `prod`, `ptp`, `put`, `ravel`, `real`, `repeat`, `reshape`, `round`, `searchsorted`, `sort`, `squeeze`, `std`, `sum`, `swapaxes`, `take`, `trace`, `transpose`, `var`.

Array conversion

`ndarray.item`(*args)	Copy an element of an array to a standard Python scalar and return it.
`ndarray.tolist`()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
`ndarray.itemset`(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
`ndarray.tostring`([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
`ndarray.tobytes`([order])	Construct Python bytes containing the raw data bytes in the array.
`ndarray.tofile`(fid[, sep, format])	Write array to a file as text or binary (default).
`ndarray.dump`(file)	Dump a pickle of the array to the specified file.
`ndarray.dumps`()	Returns the pickle of the array as a string.
`ndarray.astype`(dtype[, order, casting, ...])	Copy of the array, cast to a specified type.
`ndarray.byteswap`([inplace])	Swap the bytes of the array elements
`ndarray.copy`([order])	Return a copy of the array.
`ndarray.view`([dtype][, type])	New view of array with the same data.
`ndarray.getfield`(dtype[, offset])	Returns a field of the given array as a certain type.
`ndarray.setflags`([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

`ndarray.fill`(value)	Fill the array with a scalar value.

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an n-tuple.

`ndarray.reshape`(shape[, order])	Returns an array containing the same data with a new shape.
`ndarray.resize`(new_shape[, refcheck])	Change shape and size of array in-place.
`ndarray.transpose`(*axes)	Returns a view of the array with axes transposed.
`ndarray.swapaxes`(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
`ndarray.flatten`([order])	Return a copy of the array collapsed into one dimension.
`ndarray.ravel`([order])	Return a flattened array.
`ndarray.squeeze`([axis])	Remove axes of length one from *a*.

Item selection and manipulation

For array methods that take an *axis* keyword, it defaults to *None*. If axis is *None*, then the array is treated as a 1-D array. Any other value for *axis* represents the dimension along which the operation should proceed.

`ndarray.take`(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
`ndarray.put`(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
`ndarray.repeat`(repeats[, axis])	Repeat elements of an array.
`ndarray.choose`(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
`ndarray.sort`([axis, kind, order])	Sort an array in-place.
`ndarray.argsort`([axis, kind, order])	Returns the indices that would sort this array.
`ndarray.partition`(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
`ndarray.argpartition`(kth[, axis, kind, order])	Returns the indices that would partition this array.
`ndarray.searchsorted`(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
`ndarray.nonzero`()	Return the indices of the elements that are non-zero.
`ndarray.compress`(condition[, axis, out])	Return selected slices of this array along given axis.
`ndarray.diagonal`([offset, axis1, axis2])	Return specified diagonals.

Calculation

Many of these methods take an argument named *axis*. In such cases,

- If *axis* is *None* (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array

scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)

- If *axis* is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

Example of the *axis* argument

A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

```
>>> x = np.arange(27).reshape((3,3,3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],
       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> x.sum(axis=0)
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
>>> # for sum, axis is the first keyword, so we may omit it,
>>> # specifying only its value
>>> x.sum(0), x.sum(1), x.sum(2)
(array([[27, 30, 33],
        [36, 39, 42],
        [45, 48, 51]]),
 array([[ 9, 12, 15],
        [36, 39, 42],
        [63, 66, 69]]),
 array([[ 3, 12, 21],
        [30, 39, 48],
        [57, 66, 75]]))
```

The parameter *dtype* specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of *self*. To avoid overflow, it can be useful to perform the reduction using a larger data type.

For several methods, an optional *out* argument can also be provided and the result will be placed into the output array given. The *out* argument must be an **ndarray** and have the same number of elements. It can have a different data type in which case casting will be performed.

ndarray.max([axis, out, keepdims, initial, ...])	Return the maximum along a given axis.
ndarray.argmax([axis, out, keepdims])	Return indices of the maximum values along the given axis.
ndarray.min([axis, out, keepdims, initial, ...])	Return the minimum along a given axis.
ndarray.argmin([axis, out, keepdims])	Return indices of the minimum values along the given axis.
ndarray.ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
ndarray.clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
ndarray.conj()	Complex-conjugate all elements.
ndarray.round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
ndarray.trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.

`ndarray.sum`([axis, dtype, out, keepdims, ...])	Return the sum of the array elements over the given axis.
`ndarray.cumsum`([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
`ndarray.mean`([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
`ndarray.var`([axis, dtype, out, ddof, ...])	Returns the variance of the array elements, along given axis.
`ndarray.std`([axis, dtype, out, ddof, ...])	Returns the standard deviation of the array elements along given axis.
`ndarray.prod`([axis, dtype, out, keepdims, ...])	Return the product of the array elements over the given axis
`ndarray.cumprod`([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
`ndarray.all`([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
`ndarray.any`([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.

Arithmetic, matrix multiplication, and comparison operations

Arithmetic and comparison operations on `ndarrays` are defined as element-wise operations, and generally yield `ndarray` objects as results.

Each of the arithmetic operations (`+`, `-`, `*`, `/`, `//`, `%`, `divmod()`, `**` or `pow()`, `<<`, `>>`, `&`, `^`, `|`, `~`) and the comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) is equivalent to the corresponding universal function (or ufunc for short) in NumPy. For more information, see the section on Universal Functions.

Comparison operators:

`ndarray.__lt__`(value, /)	Return self<value.
`ndarray.__le__`(value, /)	Return self<=value.
`ndarray.__gt__`(value, /)	Return self>value.
`ndarray.__ge__`(value, /)	Return self>=value.
`ndarray.__eq__`(value, /)	Return self==value.
`ndarray.__ne__`(value, /)	Return self!=value.

Truth value of an array (`bool()`):

`ndarray.__bool__`(/)	True if self else False

Note

Truth-value testing of an array invokes `ndarray.__bool__`, which raises an error if the number of elements in the array is larger than 1, because the truth value of such arrays is ambiguous. Use `.any()` and `.all()` instead to be clear about what is meant in such cases. (If the number of elements is 0, the array evaluates to `False`.)

Unary operations:

ndarray.__neg__(/)	-self
ndarray.__pos__(/)	+self
ndarray.__abs__(self)	
ndarray.__invert__(/)	~self

Arithmetic:

ndarray.__add__(value, /)	Return self+value.
ndarray.__sub__(value, /)	Return self-value.
ndarray.__mul__(value, /)	Return self*value.
ndarray.__truediv__(value, /)	Return self/value.
ndarray.__floordiv__(value, /)	Return self//value.
ndarray.__mod__(value, /)	Return self%value.
ndarray.__divmod__(value, /)	Return divmod(self, value).
ndarray.__pow__(value[, mod])	Return pow(self, value, mod).
ndarray.__lshift__(value, /)	Return self<<value.
ndarray.__rshift__(value, /)	Return self>>value.
ndarray.__and__(value, /)	Return self&value.
ndarray.__or__(value, /)	Return self|value.
ndarray.__xor__(value, /)	Return self^value.

Note

- Any third argument to pow is silently ignored, as the underlying ufunc takes only two arguments.
- Because ndarray is a built-in type (written in C), the `__r{op}__` special methods are not directly defined.
- The functions called to implement many arithmetic special methods for arrays can be modified using __array_ufunc__.

Arithmetic, in-place:

ndarray.__iadd__(value, /)	Return self+=value.
ndarray.__isub__(value, /)	Return self-=value.
ndarray.__imul__(value, /)	Return self*=value.
ndarray.__itruediv__(value, /)	Return self/=value.
ndarray.__ifloordiv__(value, /)	Return self//=value.
ndarray.__imod__(value, /)	Return self%=value.
ndarray.__ipow__(value, /)	Return self**=value.

`ndarray.__ilshift__`(value, /)	Return self<<=value.
`ndarray.__irshift__`(value, /)	Return self>>=value.
`ndarray.__iand__`(value, /)	Return self&=value.
`ndarray.__ior__`(value, /)	Return self|=value.
`ndarray.__ixor__`(value, /)	Return self^=value.

Warning

In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, `A {op}= B` can be different than `A = A {op} B`. For example, suppose `a = ones((3,3))`. Then, `a += 3j` is different than `a = a + 3j`: while they both perform the same computation, `a += 3` casts the result to fit back in `a`, whereas `a = a + 3j` re-binds the name `a` to the result.

Matrix Multiplication:

`ndarray.__matmul__`(value, /)	Return self@value.

Note

Matrix operators `@` and `@=` were introduced in Python 3.5 following PEP 465, and the `@` operator has been introduced in NumPy 1.10.0. Further information can be found in the `matmul` documentation.

Special methods

For standard library functions:

`ndarray.__copy__`()	Used if `copy.copy` is called on an array.
`ndarray.__deepcopy__`(memo, /)	Used if `copy.deepcopy` is called on an array.
`ndarray.__reduce__`()	For pickling.
`ndarray.__setstate__`(state, /)	For unpickling.

Basic customization:

`ndarray.__new__`(*args, **kwargs)	
`ndarray.__array__`([dtype], /)	Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.
`ndarray.__array_wrap__`(array[, context], /)	Returns a view of `array` with the same type as self.

Container customization: (see Indexing)

`ndarray.__len__`(/)	Return len(self).
`ndarray.__getitem__`(key, /)	Return self[key].
`ndarray.__setitem__`(key, value, /)	Set self[key] to value.
`ndarray.__contains__`(key, /)	Return key in self.

Conversion; the operations `int()`, `float()` and `complex()`. They work only on arrays that have one element in them and return the appropriate scalar.

`ndarray.__int__`(self)	
`ndarray.__float__`(self)	
`ndarray.__complex__`	

String representations:

`ndarray.__str__`(/)	Return str(self).
`ndarray.__repr__`(/)	Return repr(self).

Utility method for typing:

`ndarray.__class_getitem__`(item, /)	Return a parametrized wrapper around the `ndarray` type.

Previous
Array objects
Next
numpy.ndarray

numpy.concatenate

numpy.concatenate(*(a1, a2, ...), axis=0, out=None, dtype=None, casting="same_kind"*)

Join a sequence of arrays along an existing axis.

Parameters: a1, a2, ... : *sequence of array_like*

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : *int, optional*

The axis along which the arrays will be joined. If axis is None, arrays are flattened before use. Default is 0.

out : *ndarray, optional*

If provided, the destination to place the result. The shape must be correct, matching that of what concatenate would have returned if no out argument were specified.

dtype : *str or dtype*

If provided, the destination array will have this dtype. Cannot be provided together with *out*.

New in version 1.20.0.

casting : *{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*

Controls what kind of data casting may occur. Defaults to 'same_kind'.

New in version 1.20.0.

Returns: res : *ndarray*

The concatenated array.

See also

ma.concatenate
Concatenate function that preserves input masks.

array_split
Split an array into multiple sub-arrays of equal or near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

hsplit
Split array into multiple sub-arrays horizontally (column wise).

vsplit
Split array into multiple sub-arrays vertically (row wise).

dsplit
Split array into multiple sub-arrays along the 3rd axis (depth).

stack
Stack a sequence of arrays along a new axis.

block
Assemble arrays from blocks.

hstack
Stack arrays in sequence horizontally (column wise).

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

column_stack
Stack 1-D arrays as columns into a 2-D array.

Notes

When one or more of the arrays to be concatenated is a MaskedArray, this function will return a MaskedArray object instead of an ndarray, but the input masks are *not* preserved. In cases where a MaskedArray is expected as input, use the ma.concatenate function from the masked array module instead.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
>>> np.concatenate((a, b), axis=None)
array([1, 2, 3, 4, 5, 6])
```

This function will not preserve masking of MaskedArray inputs.

```
>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data=[0, --, 2],
             mask=[False,  True, False],
       fill_value=999999)
>>> b
array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data=[0, 1, 2, 2, 3, 4],
             mask=False,
       fill_value=999999)
>>> np.ma.concatenate([a, b])
masked_array(data=[0, --, 2, 2, 3, 4],
             mask=[False,  True, False, False, False, False],
       fill_value=999999)
```

Previous
numpy.require

Next
numpy.stack

numpy.expand_dims

numpy.expand_dims(*a, axis*) [source]

Expand the shape of an array.

Insert a new axis that will appear at the *axis* position in the expanded array shape.

Parameters: **a** : *array_like*

Input array.

axis : *int or tuple of ints*

Position in the expanded axes where the new axis (or axes) is placed.

> ***Deprecated since version 1.13.0:*** Passing an axis where `axis > a.ndim` will be treated as `axis == a.ndim`, and passing `axis < -a.ndim - 1` will be treated as `axis == 0`. This behavior is deprecated.

> ***Changed in version 1.18.0:*** A tuple of axes is now supported. Out of range axes as described above are now forbidden and raise an **AxisError**.

Returns: **result** : *ndarray*

View of *a* with the number of dimensions increased.

See also

squeeze
The inverse operation, removing singleton dimensions
reshape
Insert, remove, and combine dimensions, and resize existing ones
doc.indexing, **atleast_1d**, **atleast_2d**, **atleast_3d**

Examples

```
>>> x = np.array([1, 2])
>>> x.shape
(2,)
```

The following is equivalent to `x[np.newaxis, :]` or `x[np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)
```

The following is equivalent to `x[:, np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=1)
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

`axis` may also be a tuple:

```
>>> y = np.expand_dims(x, axis=(0, 1))
>>> y
array([[[1, 2]]])
```

```
>>> y = np.expand_dims(x, axis=(2, 0))
>>> y
array([[[1],
        [2]]])
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```
>>> np.newaxis is None
True
```

Previous
numpy.broadcast_arrays

Next
numpy.squeeze

Search the docs ...

Array objects
Array API Standard Compatibility
Constants
Universal functions (ufunc)
Routines
Array creation routines
numpy.empty
numpy.empty_like
numpy.eye
numpy.identity
numpy.ones
numpy.ones_like
numpy.zeros
numpy.zeros_like
numpy.full
numpy.full_like
numpy.array
numpy.asarray
numpy.asanyarray
numpy.ascontiguousarray
numpy.asmatrix
numpy.copy
numpy.frombuffer
numpy.from_dlpack
numpy.fromfile
numpy.fromfunction
numpy.fromiter
numpy.fromstring
numpy.loadtxt
numpy.core.records.array
numpy.core.records.froma
numpy.core.records.fromr
numpy.core.records.froms
numpy.core.records.fromf
numpy.core.defchararray.a
numpy.core.defchararray.a
numpy.arange
numpy.linspace
numpy.logspace
numpy.geomspace
numpy.meshgrid
numpy.mgrid
numpy.ogrid
numpy.diag

numpy.linspace

numpy.linspace(*start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0*) [source]

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

> ***Changed in version 1.16.0:*** Non-scalar *start* and *stop* are now supported.

> ***Changed in version 1.20.0:*** Values are rounded towards `-inf` instead of `0` when an integer `dtype` is specified. The old behavior can still be obtained with `np.linspace(start, stop, num).astype(int)`

Parameters:

start : *array_like*

The starting value of the sequence.

stop : *array_like*

The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : *int, optional*

Number of samples to generate. Default is 50. Must be non-negative.

endpoint : *bool, optional*

If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

retstep : *bool, optional*

If True, return (*samples*, *step*), where *step* is the spacing between samples.

dtype : *dtype, optional*

The type of the output array. If **dtype** is not given, the data type is inferred from *start* and *stop*. The inferred dtype will never be an integer; *float* is chosen even if the arguments would produce an array of integers.

> ***New in version 1.9.0.***

axis : *int, optional*

The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

> ***New in version 1.16.0.***

Returns: **samples** : *ndarray*

There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

step : *float, optional*

Only returned if *retstep* is True

Size of spacing between samples.

See also

`arange`

Similar to `linspace`, but uses a step size (instead of the number of samples).

`geomspace`

Similar to `linspace`, but with numbers spaced evenly on a log scale (a geometric progression).

`logspace`

Similar to `geomspace`, but with the end points specified as logarithms.

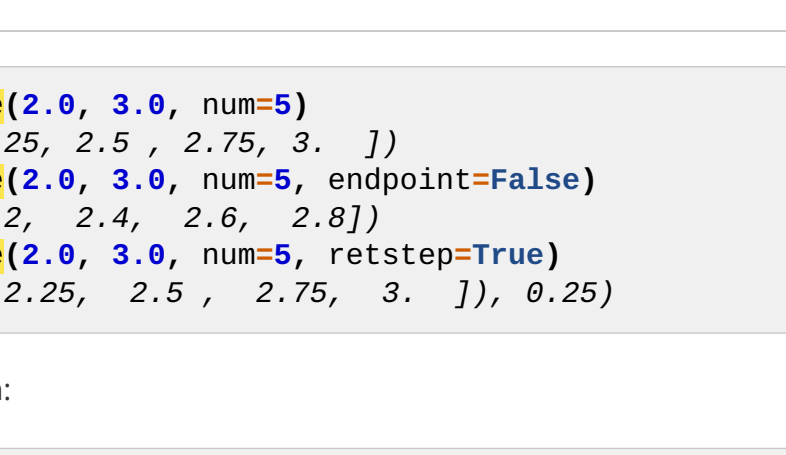
How to create arrays with regularly-spaced values

Examples

```
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```



Previous numpy.arange

Next numpy.logspace

Search the docs ...

Array objects
Array API Standard Compatibility
Constants
Universal functions (ufunc)
Routines
Array creation routines
numpy.empty
numpy.empty_like
numpy.eye
numpy.identity
numpy.ones
numpy.ones_like
numpy.zeros
numpy.zeros_like
numpy.full
numpy.full_like
numpy.array
numpy.asarray
numpy.asanyarray
numpy.ascontiguousarray
numpy.asmatrix
numpy.copy
numpy.frombuffer
numpy.from_dlpack
numpy.fromfile
numpy.fromfunction
numpy.fromiter
numpy.fromstring
numpy.loadtxt
numpy.core.records.array
numpy.core.records.froma
numpy.core.records.fromr
numpy.core.records.froms
numpy.core.records.fromf
numpy.core.defchararray.a
numpy.core.defchararray.a
numpy.arange
numpy.linspace
numpy.logspace
numpy.geomspace
numpy.meshgrid
numpy.mgrid
numpy.ogrid
numpy.diag

numpy.loadtxt

numpy.loadtxt(*fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max_rows=None, *, quotechar=None, like=None*) [source]

Load data from a text file.

Parameters: **fname** : *file, str, pathlib.Path, list of str, generator*

File, filename, list, or generator to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed. Note that generators must return bytes or strings. The strings in a list or produced by a generator are treated as lines.

dtype : *data-type, optional*

Data-type of the resulting array; default: float. If this is a structured data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.

comments : *str or sequence of str or None, optional*

The characters or list of characters used to indicate the start of a comment. None implies no comments. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is '#'.

delimiter : *str, optional*

The character used to separate the values. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is whitespace.

> *Changed in version 1.23.0:* Only single character delimiters are supported. Newline characters cannot be used as the delimiter.

converters : *dict or callable, optional*

Converter functions to customize value parsing. If *converters* is callable, the function is applied to all columns, else it must be a dict that maps column number to a parser function. See examples for further details. Default: None.

> *Changed in version 1.23.0:* The ability to pass a single callable to be applied to all columns was added.

skiprows : *int, optional*

Skip the first *skiprows* lines, including comments; default: 0.

usecols : *int or sequence, optional*

Which columns to read, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.

> *Changed in version 1.11.0:* When a single column has to be read it is possible to use an integer instead of a tuple. E.g `usecols = 3` reads the fourth column the same way as `usecols = (3,)` would.

unpack : *bool, optional*

If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`. When used with a structured data-type, arrays are returned for each field. Default is False.

ndmin : *int, optional*

The returned array will have at least *ndmin* dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2.

> *New in version 1.6.0.*

encoding : *str, optional*

Encoding used to decode the inputfile. Does not apply to input streams. The special value 'bytes' enables backward compatibility workarounds that

ensures you receive byte arrays as results if possible and passes 'latin1' encoded strings to converters. Override this value to receive unicode arrays and pass strings as input to converters. If set to None the system default is used. The default value is 'bytes'.

New in version 1.14.0.

max_rows : *int, optional*

Read *max_rows* rows of content after *skiprows* lines. The default is to read all the rows. Note that empty rows containing no data such as empty lines and comment lines are not counted towards *max_rows*, while such lines are counted in *skiprows*.

New in version 1.16.0.

Changed in version 1.23.0: Lines containing no data, including comment lines (e.g., lines starting with '#' or as specified via *comments*) are not counted towards *max_rows*.

quotechar : *unicode character or None, optional*

The character used to denote the start and end of a quoted item. Occurrences of the delimiter or comment characters are ignored within a quoted item. The default value is `quotechar=None`, which means quoting support is disabled.
If two consecutive instances of *quotechar* are found within a quoted field, the first is treated as an escape character. See examples.

New in version 1.23.0.

like : *array_like, optional*

Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

New in version 1.20.0.

Returns: out : *ndarray*

Data read from the text file.

See also

load, **fromstring**, **fromregex**

genfromtxt

Load data with missing values handled as specified.

scipy.io.loadmat

reads MATLAB data files

Notes

This function aims to be a fast reader for simply formatted files. The **genfromtxt** function provides more sophisticated handling of, e.g., lines with missing values.

Each row in the input text file must have the same number of values to be able to read all values. If all rows do not have same number of values, a subset of up to n columns (where n is the least number of values present in all rows) can be read by specifying the columns via *usecols*.

New in version 1.10.0.

The strings produced by the Python float.hex method can be used as input for floats.

Examples

```
>>> from io import StringIO   # StringIO behaves like a file object
>>> c = StringIO("0 1\n2 3")
>>> np.loadtxt(c)
array([[0., 1.],
       [2., 3.]])
```

```
>>> d = StringIO("M 21 72\nF 35 58")
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
...                      'formats': ('S1', 'i4', 'f4')})
array([(b'M', 21, 72.), (b'F', 35, 58.)],
      dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])
```

```
>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([1., 3.])
>>> y
array([2., 4.])
```

The *converters* argument is used to specify functions to preprocess the text prior to parsing. *converters* can be a dictionary that maps preprocessing functions to each column:

```
>>> s = StringIO("1.618, 2.296\n3.141, 4.669\n")
>>> conv = {
...     0: lambda x: np.floor(float(x)),  # conversion fn for column 0
...     1: lambda x: np.ceil(float(x)),  # conversion fn for column 1
... }
>>> np.loadtxt(s, delimiter=",", converters=conv)
array([[1., 3.],
       [3., 5.]])
```

converters can be a callable instead of a dictionary, in which case it is applied to all columns:

```
>>> s = StringIO("0xDE 0xAD\n0xC0 0xDE")
>>> import functools
>>> conv = functools.partial(int, base=16)
>>> np.loadtxt(s, converters=conv)
array([[222., 173.],
       [192., 222.]])
```

This example shows how *converters* can be used to convert a field with a trailing minus sign into a negative number.

```
>>> s = StringIO('10.01 31.25-\n19.22 64.31\n17.57- 63.94')
>>> def conv(fld):
...     return -float(fld[:-1]) if fld.endswith(b'-') else float(fld)
...
>>> np.loadtxt(s, converters=conv)
array([[ 10.01, -31.25],
       [ 19.22,  64.31],
       [-17.57,  63.94]])
```

Using a callable as the converter can be particularly useful for handling values with different formatting, e.g. floats with underscores:

```
>>> s = StringIO("1 2.7 100_000")
>>> np.loadtxt(s, converters=float)
array([1.e+00, 2.7e+00, 1.e+05])
```

This idea can be extended to automatically handle values specified in many different formats:

Previous numpy.fromstring
Next numpy.core.records.array

```
>>> def conv(val):
...     try:
...         return float(val)
...     except ValueError:
...         return float.fromhex(val)
>>> s = StringIO("1, 2.5, 3_000, 0b4, 0x1.4000000000000p+2")
>>> np.loadtxt(s, delimiter=",", converters=conv, encoding=None)
array([1.0e+00, 2.5e+00, 3.0e+03, 1.8e+02, 5.0e+00])
```

Note that with the default `encoding="bytes"`, the inputs to the converter function are latin-1 encoded byte strings. To deactivate the implicit encoding prior to conversion, use `encoding=None`

```
>>> s = StringIO('10.01 31.25-\n19.22 64.31\n17.57- 63.94')
>>> conv = lambda x: -float(x[:-1]) if x.endswith('-') else float(x)
>>> np.loadtxt(s, converters=conv, encoding=None)
array([[ 10.01, -31.25],
       [ 19.22,  64.31],
       [-17.57,  63.94]])
```

Support for quoted fields is enabled with the *quotechar* parameter. Comment and delimiter characters are ignored when they appear within a quoted item delineated by *quotechar*:

```
>>> s = StringIO('"alpha, #42", 10.0\n"beta, #64", 2.0\n')
>>> dtype = np.dtype([("label", "U12"), ("value", float)])
>>> np.loadtxt(s, dtype=dtype, delimiter=",", quotechar='"')
array([('alpha, #42', 10.), ('beta, #64',  2.)],
      dtype=[('label', '<U12'), ('value', '<f8')])
```

Quoted fields can be separated by multiple whitespace characters:

```
>>> s = StringIO('"alpha, #42"       10.0\n"beta, #64" 2.0\n')
>>> dtype = np.dtype([("label", "U12"), ("value", float)])
>>> np.loadtxt(s, dtype=dtype, delimiter=None, quotechar='"')
array([('alpha, #42', 10.), ('beta, #64',  2.)],
      dtype=[('label', '<U12'), ('value', '<f8')])
```

Two consecutive quote characters within a quoted field are treated as a single escaped character:

```
>>> s = StringIO('"Hello, my name is ""Monty""!"')
>>> np.loadtxt(s, dtype="U", delimiter=",", quotechar='"')
array('Hello, my name is "Monty"!', dtype='<U26')
```

Read subset of columns when all rows do not contain equal number of values:

```
>>> d = StringIO("1 2\n2 4\n3 9 12\n4 16 20")
>>> np.loadtxt(d, usecols=(0, 1))
array([[ 1.,  2.],
       [ 2.,  4.],
       [ 3.,  9.],
       [ 4., 16.]])
```

numpy.mean

numpy.mean(*a, axis=None, dtype=None, out=None, keepdims=<no value>, *, where=<no value>*) [source]

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

Parameters: **a** : *array_like*
Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis : *None or int or tuple of ints, optional*
Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

dtype : *data-type, optional*
Type to use in computing the mean. For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

out : *ndarray, optional*
Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See Output type determination for more details.

keepdims : *bool, optional*
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
If the default value is passed, then *keepdims* will not be passed through to the `mean` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

where : *array_like of bool, optional*
Elements to include in the mean. See `reduce` for details.

New in version 1.20.0.

Returns: **m** : *ndarray, see dtype parameter above*
If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See also

`average`
Weighted average
`std`, `var`, `nanmean`, `nanstd`, `nanvar`

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see

example below). Specifying a higher-precision accumulator using the **dtype** keyword can alleviate this issue.

By default, **float16** results are computed using **float32** intermediates for extra precision.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

In single precision, **mean** can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary
```

Specifying a where argument:

```
>>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
>>> np.mean(a)
12.0
>>> np.mean(a, where=[[True], [False], [False]])
9.0
```

Previous
numpy.average

Next
numpy.std

numpy.meshgrid

numpy.meshgrid(**xi, copy=True, sparse=False, indexing='xy'*) [source]

Return a list of coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

> ***Changed in version 1.9:*** 1-D and 0-D cases are allowed.

Parameters: **x1, x2,..., xn** : *array_like*

1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*

Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.

> ***New in version 1.7.0.***

sparse : *bool, optional*

If True the shape of the returned coordinate array for dimension *i* is reduced from `(N1, ..., Ni, ... Nn)` to `(1, ..., 1, Ni, 1, ..., 1)`. These sparse coordinate grids are intended to be use with Broadcasting. When all coordinates are used in an expression, broadcasting still leads to a fully-dimensonal result array. Default is False.

> ***New in version 1.7.0.***

copy : *bool, optional*

If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.

> ***New in version 1.7.0.***

Returns: **X1, X2,..., XN** : *list of ndarrays*

For vectors *x1*, *x2*,..., *xn* with lengths `Ni=len(xi)`, returns `(N1, N2, N3,..., Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,..., Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

See also

mgrid
Construct a multi-dimensional "meshgrid" using indexing notation.
ogrid
Construct an open multi-dimensional "meshgrid" using indexing notation.
how-to-index

Notes

This function supports both indexing conventions through the indexing keyword argument. Giving the string 'ij' returns a meshgrid with matrix indexing, while 'xy' returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing. The difference is illustrated by the following code snippet:

```
xv, yv = np.meshgrid(x, y, indexing='ij')
for i in range(nx):
    for j in range(ny):
        # treat xv[i,j], yv[i,j]

xv, yv = np.meshgrid(x, y, indexing='xy')
for i in range(nx):
    for j in range(ny):
        # treat xv[j,i], yv[j,i]
```

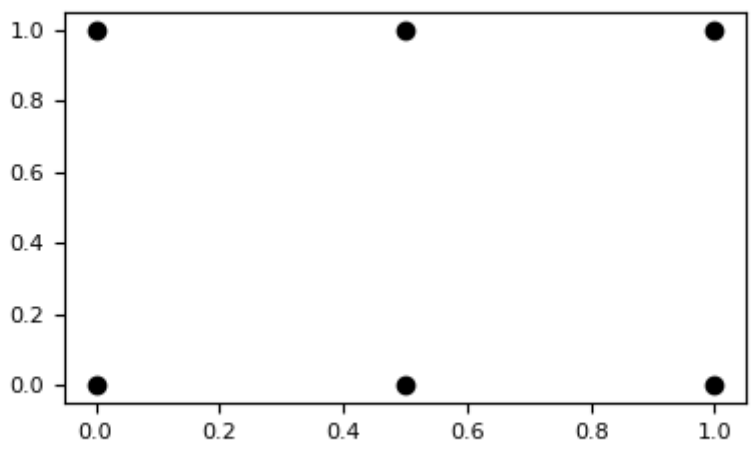
In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

Examples

```
>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = np.meshgrid(x, y)
>>> xv
array([[0. , 0.5, 1. ],
       [0. , 0.5, 1. ]])
>>> yv
array([[0.,  0.,  0.],
       [1.,  1.,  1.]])
```

The result of `meshgrid` is a coordinate grid:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(xv, yv, marker='o', color='k', linestyle='none')
>>> plt.show()
```



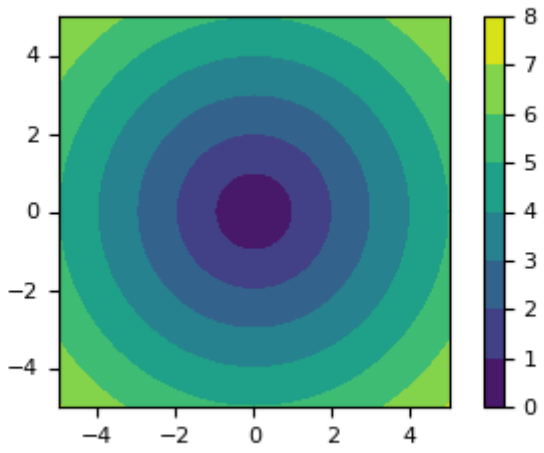
You can create sparse output arrays to save memory and computation time.

```
>>> xv, yv = np.meshgrid(x, y, sparse=True)
>>> xv
array([[0. ,  0.5,  1. ]])
>>> yv
array([[0.],
       [1.]])
```

`meshgrid` is very useful to evaluate functions on a grid. If the function depends on all coordinates, both dense and sparse outputs can be used.

```
>>> x = np.linspace(-5, 5, 101)
>>> y = np.linspace(-5, 5, 101)
>>> # full coordinate arrays
>>> xx, yy = np.meshgrid(x, y)
>>> zz = np.sqrt(xx**2 + yy**2)
>>> xx.shape, yy.shape, zz.shape
((101, 101), (101, 101), (101, 101))
>>> # sparse coordinate arrays
>>> xs, ys = np.meshgrid(x, y, sparse=True)
>>> zs = np.sqrt(xs**2 + ys**2)
>>> xs.shape, ys.shape, zs.shape
((1, 101), (101, 1), (101, 101))
>>> np.array_equal(zz, zs)
True
```

```
>>> h = plt.contourf(x, y, zs)
>>> plt.axis('scaled')
>>> plt.colorbar()
>>> plt.show()
```



Previous
numpy.geomspace

Next
numpy.mgrid

numpy.random.random

`random.random(size=None)`

Return random floats in the half-open interval [0.0, 1.0). Alias for `random_sample` to ease forward-porting to the new random API.

Created using Sphinx 5.3.0.

numpy.random.seed

random.seed(*seed=None*)

Reseed the singleton RandomState instance.

> **See also**
>
> numpy.random.Generator

Notes

This is a convenience, legacy function that exists to support older code that uses the singleton RandomState. Best practice is to use a dedicated `Generator` instance rather than the random variate generation methods exposed directly in the random module.

Previous numpy.random.sample

Next numpy.random.set_state

Created using Sphinx 5.3.0.

numpy.random.uniform

random.uniform(*low=0.0, high=1.0, size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by **uniform**.

Note

New code should use the **uniform** method of a **Generator** instance instead; please see the Quick Start.

Parameters: low : *float or array_like of floats, optional*

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high : *float or array_like of floats*

Upper boundary of the output interval. All values generated will be less than or equal to high. The high limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. The default value is 1.0.

size : *int or tuple of ints, optional*

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns: out : *ndarray or scalar*

Drawn samples from the parameterized uniform distribution.

See also

randint
Discrete uniform distribution, yielding integers.

random_integers
Discrete uniform distribution over the closed interval `[low, high]`.

random_sample
Floats uniformly distributed over `[0, 1)`.

random
Alias for **random_sample**.

rand
Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

random.Generator.uniform
which should be used for new code.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition. The `high` limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. For example:

```
>>> x = np.float32(5*0.99999999)
>>> x
5.0
```

Examples

Draw samples from the distribution:

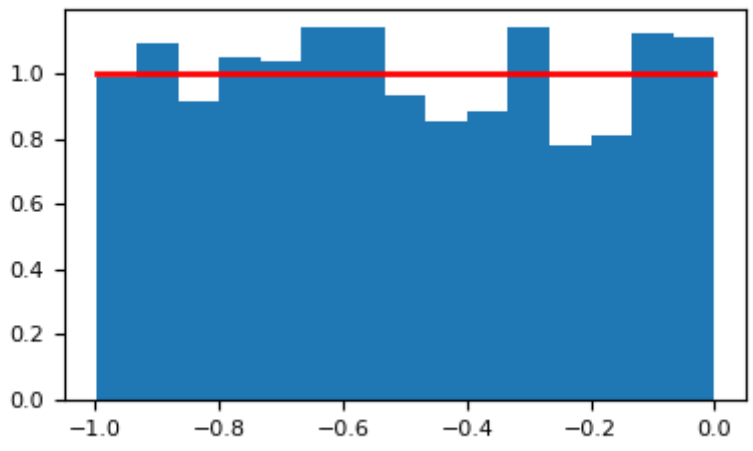
```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```



Previous
numpy.random.triangular

Next
numpy.random.vonmises

numpy.std

numpy.std(*a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>, *, where=<no value>*) [source]

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters:

a : *array_like*
Calculate the standard deviation of these values.

axis : *None or int or tuple of ints, optional*
Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

> *New in version 1.7.0.*
>
> If this is a tuple of ints, a standard deviation is performed over multiple axes, instead of a single axis or all the axes as before.

dtype : *dtype, optional*
Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

out : *ndarray, optional*
Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

ddof : *int, optional*
Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims : *bool, optional*
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
If the default value is passed, then *keepdims* will not be passed through to the **std** method of sub-classes of **ndarray**, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

where : *array_like of bool, optional*
Elements to include in the standard deviation. See **reduce** for details.

> *New in version 1.20.0.*

Returns:

standard_deviation : *ndarray, see dtype parameter above.*
If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

> **See also**
>
> **var**, **mean**, **nanmean**, **nanstd**, **nanvar**
> Output type determination

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `std = sqrt(mean(x))`, where `x = abs(a - a.mean())**2`.

The average squared deviation is typically calculated as `x.sum() / N`, where `N = len(x)`. If, however,

ddof is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, **std** takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the **dtype** keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949 # may vary
>>> np.std(a, axis=0)
array([1.,  1.])
>>> np.std(a, axis=1)
array([0.5,  0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
0.45000005
```

Computing the standard deviation in float64 is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.44999999925494177 # may vary
```

Specifying a where argument:

```
>>> a = np.array([[14, 8, 11, 10], [7, 9, 10, 11], [10, 15, 5, 10]])
>>> np.std(a)
2.614064523559687 # may vary
>>> np.std(a, where=[[True], [True], [False]])
2.0
```

Previous
numpy.mean

Next
numpy.var

numpy.unique

numpy.unique(*ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *, equal_nan=True*) [source]

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

Parameters:

ar : *array_like*
Input array. Unless *axis* is specified, this will be flattened if it is not already 1-D.

return_index : *bool, optional*
If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

return_inverse : *bool, optional*
If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.

return_counts : *bool, optional*
If True, also return the number of times each unique item appears in *ar*.

axis : *int or None, optional*
The axis to operate on. If None, *ar* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.

New in version 1.13.0.

equal_nan : *bool, optional*
If True, collapses multiple NaN values in the return array into one.

New in version 1.24.

Returns:

unique : *ndarray*
The sorted unique values.

unique_indices : *ndarray, optional*
The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

unique_inverse : *ndarray, optional*
The indices to reconstruct the original array from the unique array. Only provided if *return_inverse* is True.

unique_counts : *ndarray, optional*
The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

New in version 1.9.0.

See also

numpy.lib.arraysetops
: Module with a number of other functions for performing set operations on arrays.

repeat
: Repeat elements of an array.

Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order. The flattened subarrays are then viewed as a structured type with each element given a label, with the effect that we end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the unique rows of a 2D array

```
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'], dtype='<U1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'], dtype='<U1')
```

Reconstruct the input array from the unique values and inverse:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

Reconstruct the input values from the unique values and counts:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> values, counts = np.unique(a, return_counts=True)
>>> values
array([1, 2, 3, 4, 6])
>>> counts
array([1, 3, 1, 1, 1])
>>> np.repeat(values, counts)
array([1, 2, 2, 2, 3, 4, 6])    # original order not preserved
```

Previous
numpy.trim_zeros

Next
numpy.flip

numpy.zeros

numpy.zeros(*shape, dtype=float, order='C', *, like=None*)

Return a new array of given shape and type, filled with zeros.

Parameters:

shape : *int or tuple of ints*

Shape of the new array, e.g., `(2, 3)` or `2`.

dtype : *data-type, optional*

The desired data-type for the array, e.g., **`numpy.int8`**. Default is **`numpy.float64`**.

order : *{'C', 'F'}, optional, default: 'C'*

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

like : *array_like, optional*

Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

New in version 1.20.0.

Returns:

out : *ndarray*

Array of zeros with the given shape, dtype, and order.

See also

`zeros_like`
Return an array of zeros with shape and type of input.

`empty`
Return a new uninitialized array.

`ones`
Return a new array setting values to one.

`full`
Return a new array of given shape filled with value.

Examples

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

Previous
numpy.ones_like

Next
numpy.zeros_like

sklearn.datasets.load_iris

`sklearn.datasets.load_iris(*, return_X_y=False, as_frame=False)` [source]

Load and return the iris dataset (classification).

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Read more in the User Guide.

Parameters:

return_X_y : *bool, default=False*

If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

as_frame : *bool, default=False*

If True, the data is a pandas DataFrame including columns with appropriate dtypes (numeric). The target is a pandas DataFrame or Series depending on the number of target columns. If `return_X_y` is True, then (`data`, `target`) will be pandas DataFrames or Series as described below.

New in version 0.23.

Returns:

data : *Bunch*

Dictionary-like object, with the following attributes.

data : *{ndarray, dataframe} of shape (150, 4)*

The data matrix. If `as_frame=True`, `data` will be a pandas DataFrame.

target: {ndarray, Series} of shape (150,)

The classification target. If `as_frame=True`, `target` will be a pandas Series.

feature_names: list

The names of the dataset columns.

target_names: list

The names of target classes.

frame: DataFrame of shape (150, 5)

Only present when `as_frame=True`. DataFrame with `data` and `target`.

New in version 0.23.

DESCR: str

The full description of the dataset.

filename: str

The path to the location of the data.

New in version 0.20.

(data, target) : *tuple if `return_X_y` is True*

A tuple of two ndarray. The first containing a 2D array of shape (n_samples, n_features) with each row representing one sample and each column representing the features. The second ndarray of shape (n_samples,) containing the target samples.

New in version 0.18.

Notes

Changed in version 0.20: Fixed two wrong data points according to Fisher's paper. The new version is the same as in R, but not as in the UCI Machine Learning Repository.

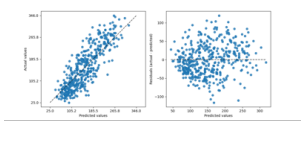
Examples

Let’s say you are interested in the samples 10, 25, and 50, and want to know their class name.

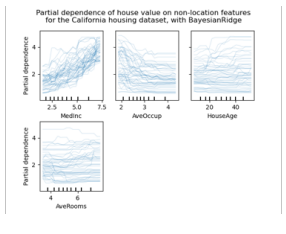
```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> data.target[[10, 25, 50]]
array([0, 0, 1])
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```

See The Iris Dataset for a more detailed example of how to work with the iris dataset.

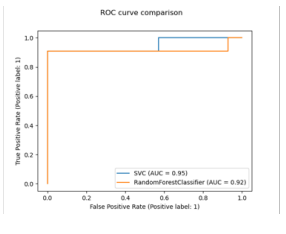
Examples using `sklearn.datasets.load_iris`



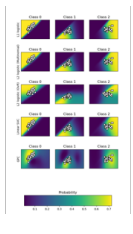
Release Highlights for scikit-learn 1.2



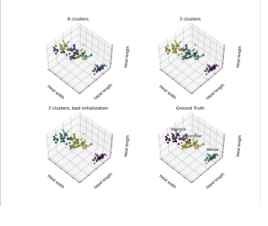
Release Highlights for scikit-learn 0.24



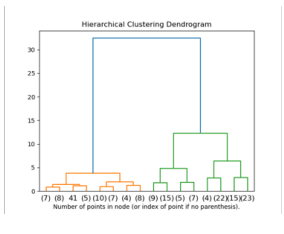
Release Highlights for scikit-learn 0.22



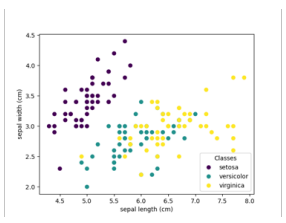
Plot classification probability



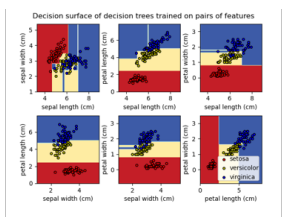
K-means Clustering



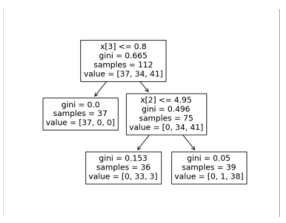
Plot Hierarchical Clustering Dendrogram



The Iris Dataset

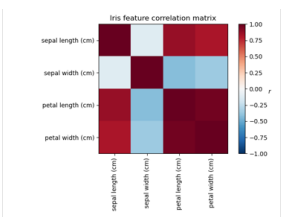


Plot the decision surface of decision trees trained on the iris dataset

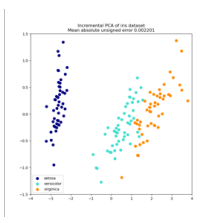


Understanding the decision tree structure

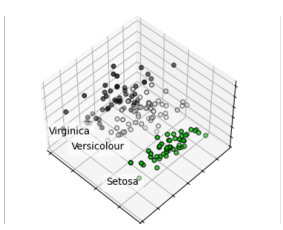
Comparison of LDA and PCA 2D projection of Iris dataset



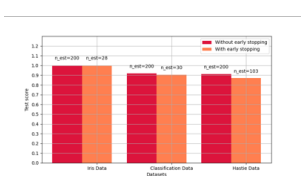
Factor Analysis (with rotation) to visualize patterns



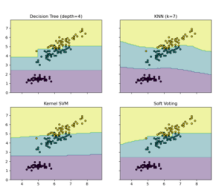
Incremental PCA



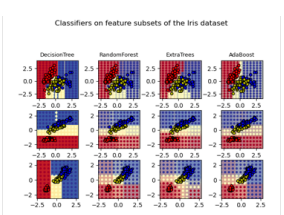
PCA example with Iris Data-set



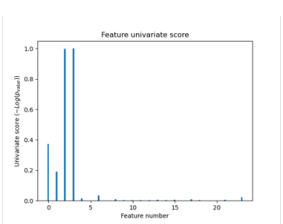
Early stopping of Gradient Boosting



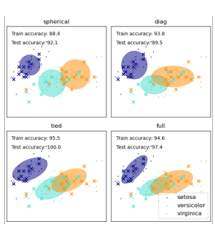
Plot the decision boundaries of a VotingClassifier



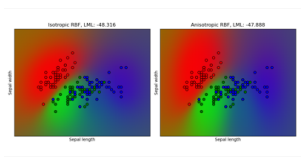
Plot the decision surfaces of ensembles of trees on the iris dataset



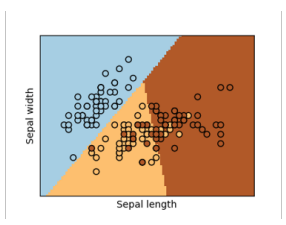
Univariate Feature Selection



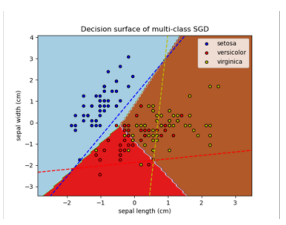
GMM covariances



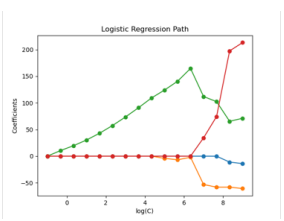
Gaussian process classification (GPC) on iris dataset



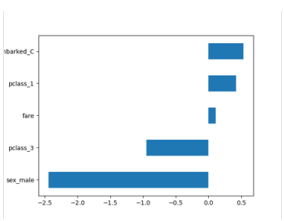
Logistic Regression 3-class Classifier



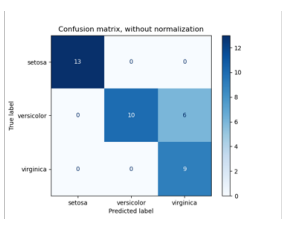
Plot multi-class SGD on the iris dataset



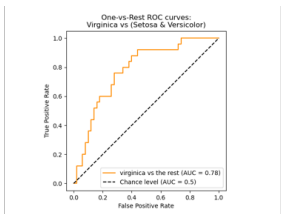
Regularization path of L1- Logistic Regression



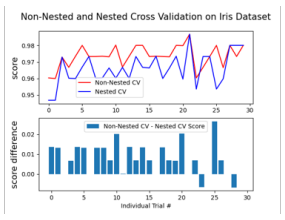
Introducing the set_output API



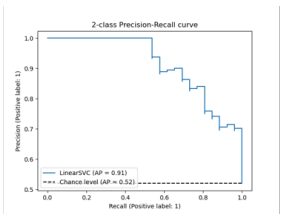
Confusion matrix



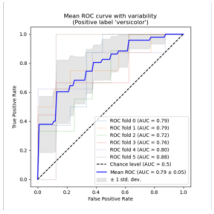
Multiclass Receiver Operating Characteristic (ROC)



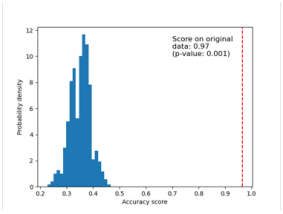
Nested versus non-nested cross-validation



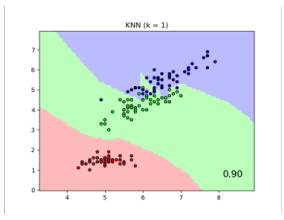
Precision-Recall



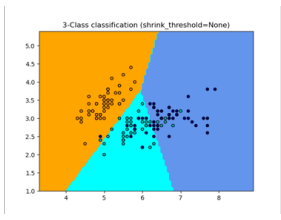
Receiver Operating Characteristic (ROC) with cross validation



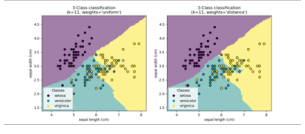
Test with permutations the significance of a classification score



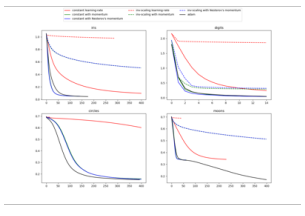
Comparing Nearest Neighbors with and without Neighborhood Components Analysis



Nearest Centroid Classification

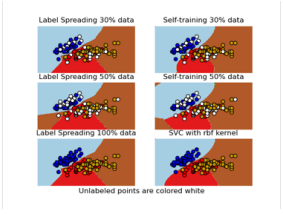


Nearest Neighbors Classification

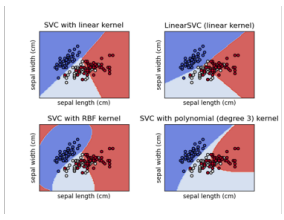


Compare Stochastic learning strategies for MLPClassifier

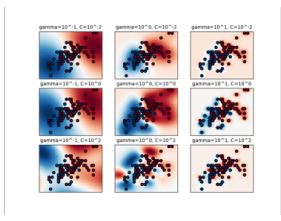
Concatenating multiple feature extraction methods



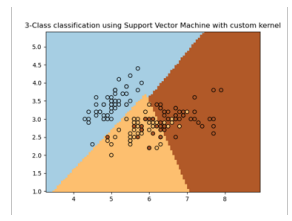
Decision boundary of semi-supervised classifiers versus SVM on the Iris dataset



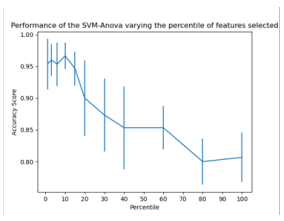
Plot different SVM classifiers in the iris dataset



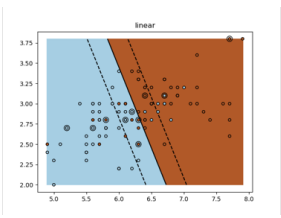
RBF SVM parameters



SVM with custom kernel



SVM-Anova: SVM with univariate feature selection



SVM Exercise

sklearn.datasets.make_moons

`sklearn.datasets.make_moons`(*n_samples=100*, ***, *shuffle=True*, *noise=None*, *random_state=None*) [source]

Make two interleaving half circles.

A simple toy dataset to visualize clustering and classification algorithms. Read more in the User Guide.

Parameters:

n_samples : *int or tuple of shape (2,), dtype=int, default=100*
If int, the total number of points generated. If two-element tuple, number of points in each of two moons.

Changed in version 0.23: Added two-element tuple.

shuffle : *bool, default=True*
Whether to shuffle the samples.

noise : *float, default=None*
Standard deviation of Gaussian noise added to the data.

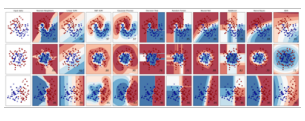
random_state : *int, RandomState instance or None, default=None*
Determines random number generation for dataset shuffling and noise. Pass an int for reproducible output across multiple function calls. See Glossary.

Returns:

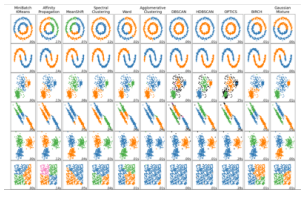
X : *ndarray of shape (n_samples, 2)*
The generated samples.

y : *ndarray of shape (n_samples,)*
The integer labels (0 or 1) for class membership of each sample.

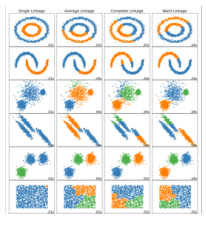
Examples using `sklearn.datasets.make_moons`



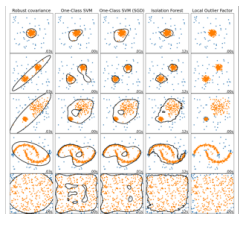
Classifier comparison



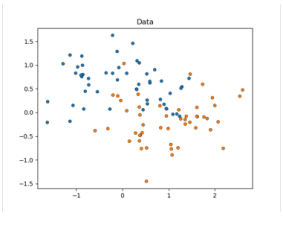
Comparing different clustering algorithms on toy datasets



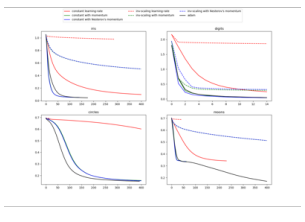
Comparing different hierarchical linkage methods on toy datasets



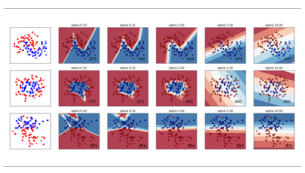
Comparing anomaly detection algorithms for outlier detection on toy datasets



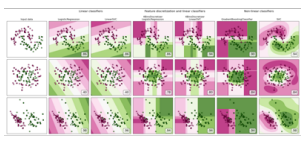
Statistical comparison of models using grid search



Compare Stochastic learning strategies for MLPClassifier



Varying regularization in Multi-layer Perceptron



Feature discretization

sklearn.tree.DecisionTreeClassifier

class `sklearn.tree.`**`DecisionTreeClassifier`**(**, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None, ccp_alpha=0.0*) [source]

A decision tree classifier.

Read more in the User Guide.

Toggle Menu

Parameters:

criterion : *{"gini", "entropy", "log_loss"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see Mathematical formulation.

splitter : *{"best", "random"}, default="best"*

The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth : *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int or float, default=2*

The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf : *int or float, default=1*

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf : *float, default=0.0*

The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features : *int, float or {"auto", "sqrt", "log2"}, default=None*

The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state : *int, RandomState instance or None, default=None*

Controls the randomness of the estimator. The features are always randomly permuted at each split, even if `splitter` is set to `"best"`. When `max_features < n_features`, the algorithm will select `max_features` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if `max_features=n_features`. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed to an integer. See Glossary for details.

max_leaf_nodes : *int, default=None*

Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease : *float, default=0.0*

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

rsion 0.19.

class_weight : *dict, list of dict or "balanced", default=None*

Weights associated with classes in the form `{class_label: weight}`. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha : *non-negative float, default=0.0*

Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See Minimal Cost-Complexity Pruning for details.

New in version 0.22.

Attributes:

classes_ : *ndarray of shape (n_classes,) or list of ndarray*

The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

`feature_importances_` : *ndarray of shape (n_features,)*

Return the feature importances.

max_features_ : *int*

The inferred value of max_features.

n_classes_ : *int or list of int*

The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

n_features_in_ : *int*

Number of features seen during fit.

New in version 0.24.

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*

Names of features seen during fit. Defined only when `X` has feature names that are all strings.

New in version 1.0.

n_outputs_ : *int*

The number of outputs when `fit` is performed.

tree_ : *Tree instance*

The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and Understanding the decision tree structure for basic usage of these attributes.

See also:

`DecisionTreeRegressor`

A decision tree regressor.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The **`predict`** method operates using the **`numpy.argmax`** function on the outputs of **`predict_proba`**. This means that in case the highest predicted probabilities are tied, the classifier will predict the tied class with the lowest index in classes_.

References

[1]
https://en.wikipedia.org/wiki/Decision_tree_learning

[2]
L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

T. Hastie, R. Tibshirani and J. Friedman. “Elements of Statistical Learning”, Springer, 2009.

[4]
L. Breiman, and A. Cutler, “Random Forests”, https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Methods

`apply`(X[, check_input])	Return the index of the leaf that each sample is predicted as.
`cost_complexity_pruning_path`(X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
`decision_path`(X[, check_input])	Return the decision path in the tree.
`fit`(X, y[, sample_weight, check_input])	Build a decision tree classifier from the training set (X, y).
`get_depth`()	Return the depth of the decision tree.
`get_metadata_routing`()	Get metadata routing of this object.
`get_n_leaves`()	Return the number of leaves of the decision tree.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, check_input])	Predict class or regression value for X.
`predict_log_proba`(X)	Predict class log-probabilities of the input samples X.
`predict_proba`(X[, check_input])	Predict class probabilities of the input samples X.
`score`(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
`set_fit_request`(*[, check_input, sample_weight])	Request metadata passed to the `fit` method.
`set_params`(**params)	Set the parameters of this estimator.
`set_predict_proba_request`(*[, check_input])	Request metadata passed to the `predict_proba` method.
`set_predict_request`(*[, check_input])	Request metadata passed to the `predict` method.
`set_score_request`(*[, sample_weight])	Request metadata passed to the `score` method.

apply(*X*, *check_input=True*) [source]

Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : *bool, default=True*
Allow to bypass several input checking. Don’t use this parameter unless you know what you’re doing.

Returns:

X_leaves : *array-like of shape (n_samples,)*
For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

cost_complexity_pruning_path(*X*, *y*, *sample_weight=None*) [source]

Compute the pruning path during Minimal Cost-Complexity Pruning.

See Minimal Cost-Complexity Pruning for details on the pruning process.

Parameters:

X : ***{array-like, sparse matrix} of shape (n_samples, n_features)***
The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y : ***array-like of shape (n_samples,) or (n_samples, n_outputs)***
The target values (class labels) as integers or strings.

sample_weight : ***array-like of shape (n_samples,), default=None***
Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns:

ccp_path : ***Bunch***
Dictionary-like object, with the following attributes.

ccp_alphas : ***ndarray***
Effective alphas of subtree during pruning.

impurities : ***ndarray***
Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

`decision_path`(*X*, *check_input=True*) [source]

Return the decision path in the tree.

New in version 0.18.

Parameters:

X : ***{array-like, sparse matrix} of shape (n_samples, n_features)***
The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : ***bool, default=True***
Allow to bypass several input checking. Don't use this parameter unless you know what you're doing.

Returns:

indicator : ***sparse matrix of shape (n_samples, n_nodes)***
Return a node indicator CSR matrix where non zero elements indicates that the samples goes through the nodes.

---

property **`feature_importances_`**
Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See `sklearn.inspection.permutation_importance` as an alternative.

Returns:

feature_importances_ : ***ndarray of shape (n_features,)***
Normalized total reduction of criteria by feature (Gini importance).

`fit`(*X*, *y*, *sample_weight=None*, *check_input=True*) [source]

Build a decision tree classifier from the training set (X, y).

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y : *array-like of shape (n_samples,) or (n_samples, n_outputs)*
The target values (class labels) as integers or strings.

sample_weight : *array-like of shape (n_samples,), default=None*
Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input : *bool, default=True*
Allow to bypass several input checking. Don't use this parameter unless you know what you're doing.

Returns:

self : *DecisionTreeClassifier*
Fitted estimator.

`get_depth`() [source]

Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns:

self.tree_.max_depth : *int*
The maximum depth of the tree.

`get_metadata_routing`() [source]

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns:

routing : *MetadataRequest*
A `MetadataRequest` encapsulating routing information.

`get_n_leaves`() [source]

Return the number of leaves of the decision tree.

Returns:

self.tree_.n_leaves : *int*
Number of leaves.

`get_params`(*deep=True*) [source]

Get parameters for this estimator.

Parameters:

deep : *bool, default=True*
If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

params : *dict*
Parameter names mapped to their values.

`predict`(*X*, *check_input=True*) [source]

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : *bool, default=True*
Allow to bypass several input checking. Don't use this parameter unless you know what you're doing.

Returns:

y : *array-like of shape (n_samples,) or (n_samples, n_outputs)*
The predicted classes, or the predict values.

`predict_log_proba`(*X*) [source]

Predict class log-probabilities of the input samples X.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns:

proba : *ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1*
The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute classes_.

`predict_proba`(*X*, *check_input=True*) [source]

Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : *bool, default=True*
Allow to bypass several input checking. Don't use this parameter unless you know what you're doing.

Returns:

proba : *ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1*
The class probabilities of the input samples. The order of the classes corresponds to that in the attribute classes_.

`score`(*X*, *y*, *sample_weight=None*) [source]

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:

X : *array-like of shape (n_samples, n_features)*
Test samples.

y : *array-like of shape (n_samples,) or (n_samples, n_outputs)*
True labels for `X`.

sample_weight : *array-like of shape (n_samples,), default=None*
Sample weights.

Returns:

score : *float*
Mean accuracy of `self.predict(X)` w.r.t. `y`.

`set_fit_request`(*, *check_input: Union[bool, None, str] = '$UNCHANGED$'*, *sample_weight: Union[bool, None, str] = '$UNCHANGED$'*) → DecisionTreeClassifier [source]

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see **`sklearn.set_config`**). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a **`Pipeline`**. Otherwise it has no effect.

Parameters:

check_input : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*
Metadata routing for `check_input` parameter in `fit`.

sample_weight : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*
Metadata routing for `sample_weight` parameter in `fit`.

Returns:

self : *object*
The updated object.

`set_params`(***params*) [source]

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as **`Pipeline`**). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters:

****params : *dict***
Estimator parameters.

Returns:

self : *estimator instance*
Estimator instance.

`set_predict_proba_request`(**, check_input: Union[bool, None, str] = '$UNCHANGED$'*) → DecisionTreeClassifier [source]

Request metadata passed to the `predict_proba` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see **`sklearn.set_config`**). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict_proba` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict_proba`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a **`Pipeline`**. Otherwise it has no effect.

Parameters:

check_input : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

Metadata routing for `check_input` parameter in `predict_proba`.

Returns:

self : *object*

The updated object.

`set_predict_request`(*, *check_input: Union[bool, None, str] = '$UNCHANGED$'*) → DecisionTreeClassifier [source]

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see **`sklearn.set_config`**). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a **`Pipeline`**. Otherwise it has no effect.

Parameters:

check_input : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

Metadata routing for `check_input` parameter in `predict`.

Returns:

self : *object*

The updated object.

`set_score_request`(*, *sample_weight: Union[bool, None, str] = '$UNCHANGED$'*) → DecisionTreeClassifier [source]

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see **`sklearn.set_config`**). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a **`Pipeline`**. Otherwise it has no effect.

Parameters:

sample_weight : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

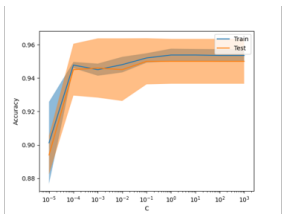
Metadata routing for `sample_weight` parameter in `score`.

Returns:

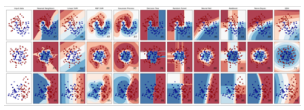
self : *object*

The updated object.

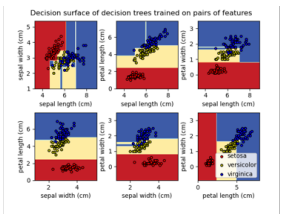
Examples using `sklearn.tree.DecisionTreeClassifier`



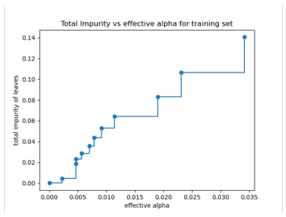
Release Highlights for scikit-learn 1.3



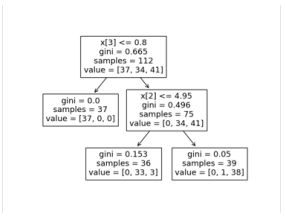
Classifier comparison



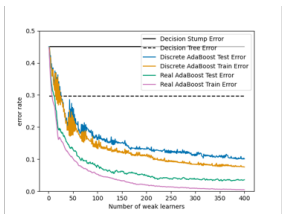
Plot the decision surface of decision trees trained on the iris dataset



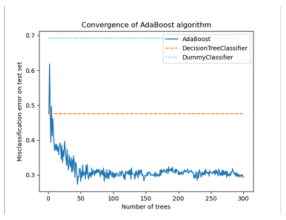
Post pruning decision trees with cost complexity pruning



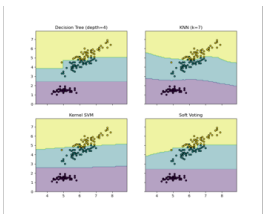
Understanding the decision tree structure



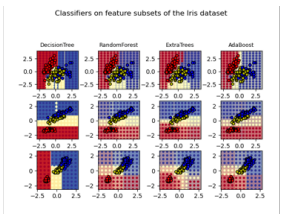
Discrete versus Real AdaBoost



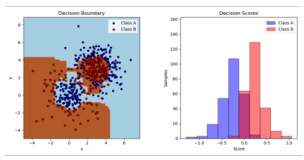
Multi-class AdaBoosted Decision Trees



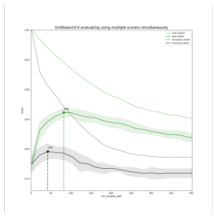
Plot the decision boundaries of a VotingClassifier



Plot the decision surfaces of ensembles of trees on the iris dataset



Two-class AdaBoost



Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV

PyTorch

Table of Contents

TORCH.TENSOR

A `torch.Tensor` is a multi-dimensional matrix containing elements of a single data type.

Data types

Torch defines 10 tensor types with CPU and GPU variants which are as follows:

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point 1	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
16-bit floating point 2	`torch.bfloat16`	`torch.BFloat16Tensor`	`torch.cuda.BFloat16Tensor`
32-bit complex	`torch.complex32` or `torch.chalf`		
64-bit complex	`torch.complex64` or `torch.cfloat`		
128-bit complex	`torch.complex128` or `torch.cdouble`		
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`
Boolean	`torch.bool`	`torch.BoolTensor`	`torch.cuda.BoolTensor`
quantized 8-bit integer (unsigned)	`torch.quint8`	`torch.ByteTensor`	/
quantized 8-bit integer (signed)	`torch.qint8`	`torch.CharTensor`	/
quantized 32-bit integer (signed)	`torch.qint32`	`torch.IntTensor`	/
quantized 4-bit integer (unsigned) 3	`torch.quint4x2`	`torch.ByteTensor`	/

1

Sometimes referred to as binary16: uses 1 sign, 5 exponent, and 10 significand bits. Useful when precision is important at the expense of range.

2

Sometimes referred to as Brain Floating Point: uses 1 sign, 8 exponent, and 7 significand bits. Useful when range is important, since it has the same number of exponent bits as `float32`

3

quantized 4-bit integer is stored as a 8-bit signed integer. Currently it's only supported in EmbeddingBag operator.

`torch.Tensor` is an alias for the default tensor type (`torch.FloatTensor`).

Initializing and basic operations

A tensor can be constructed from a Python `list` or sequence using the `torch.tensor()` constructor:

```
>>> torch.tensor([[1., -1.], [1., -1.]])
tensor([[ 1.0000, -1.0000],
        [ 1.0000, -1.0000]])
>>> torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
```

WARNING

`torch.tensor()` always copies `data` . If you have a Tensor `data` and just want to change its `requires_grad` flag, use `requires_grad_()` or `detach()` to avoid a copy. If you have a numpy array and want to avoid a copy, use `torch.as_tensor()` .

A tensor of specific data type can be constructed by passing a `torch.dtype` and/or a `torch.device` to a constructor or tensor creation op:

```
>>> torch.zeros([2, 4], dtype=torch.int32)
tensor([[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]], dtype=torch.int32)
>>> cuda0 = torch.device('cuda:0')
>>> torch.ones([2, 4], dtype=torch.float64, device=cuda0)
tensor([[ 1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000]], dtype=torch.float64, device='cuda:0')
```

For more information about building Tensors, see Creation Ops

The contents of a tensor can be accessed and modified using Python's indexing and slicing notation:

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1][2])
tensor(6)
>>> x[0][1] = 8
>>> print(x)
tensor([[ 1,  8,  3],
        [ 4,  5,  6]])
```

Use `torch.Tensor.item()` to get a Python number from a tensor containing a single value:

```
>>> x = torch.tensor([[1]])
>>> x
tensor([[ 1]])
>>> x.item()
1
>>> x = torch.tensor(2.5)
>>> x
tensor(2.5000)
>>> x.item()
2.5
```

For more information about indexing, see Indexing, Slicing, Joining, Mutating Ops

A tensor can be created with `requires_grad=True` so that `torch.autograd` records operations on them for automatic differentiation.

```
>>> x = torch.tensor([[1., -1.], [1., 1.]], requires_grad=True)
>>> out = x.pow(2).sum()
>>> out.backward()
>>> x.grad
tensor([[ 2.0000, -2.0000],
        [ 2.0000,  2.0000]])
```

Each tensor has an associated `torch.Storage` , which holds its data. The tensor class also provides multi-dimensional, strided view of a storage and defines numeric operations on it.

NOTE

For more information on tensor views, see Tensor Views.

NOTE

For more information on the `torch.dtype` , `torch.device` , and `torch.layout` attributes of a `torch.Tensor` , see Tensor Attributes.

NOTE

Methods which mutate a tensor are marked with an underscore suffix. For example, `torch.FloatTensor.abs_()` computes the absolute value in-place and returns the modified tensor, while `torch.FloatTensor.abs()` computes the result in a new tensor.

• NOTE

To change an existing tensor's `torch.device` and/or `torch.dtype`, consider using `to()` method on the tensor.

• WARNING

Current implementation of `torch.Tensor` introduces memory overhead, thus it might lead to unexpectedly high memory usage in the applications with many tiny tensors. If this is your case, consider using one large structure.

Tensor class reference

CLASS torch.Tensor

There are a few main ways to create a tensor, depending on your use case.

- To create a tensor with pre-existing data, use `torch.tensor()`.
- To create a tensor with specific size, use `torch.*` tensor creation ops (see Creation Ops).
- To create a tensor with the same size (and similar types) as another tensor, use `torch.*_like` tensor creation ops (see Creation Ops).
- To create a tensor with similar type but different size as another tensor, use `tensor.new_*` creation ops.

Tensor.T

Returns a view of this tensor with its dimensions reversed.

If `n` is the number of dimensions in `x`, `x.T` is equivalent to `x.permute(n-1, n-2, ..., 0)`.

• WARNING

The use of `Tensor.T()` on tensors of dimension other than 2 to reverse their shape is deprecated and it will throw an error in a future release. Consider `mT` to transpose batches of matrices or *x.permute(*torch.arange(x.ndim - 1, -1, -1))* to reverse the dimensions of a tensor.

Tensor.H

Returns a view of a matrix (2-D tensor) conjugated and transposed.

`x.H` is equivalent to `x.transpose(0, 1).conj()` for complex matrices and `x.transpose(0, 1)` for real matrices.

• SEE ALSO

`mH`: An attribute that also works on batches of matrices.

Tensor.mT

Returns a view of this tensor with the last two dimensions transposed.

`x.mT` is equivalent to `x.transpose(-2, -1)`.

Tensor.mH

Accessing this property is equivalent to calling `adjoint()`.

`Tensor.new_tensor`	Returns a new Tensor with `data` as the tensor data.
`Tensor.new_full`	Returns a Tensor of size `size` filled with `fill_value`.
`Tensor.new_empty`	Returns a Tensor of size `size` filled with uninitialized data.
`Tensor.new_ones`	Returns a Tensor of size `size` filled with `1`.
`Tensor.new_zeros`	Returns a Tensor of size `size` filled with `0`.
`Tensor.is_cuda`	Is `True` if the Tensor is stored on the GPU, `False` otherwise.
`Tensor.is_quantized`	Is `True` if the Tensor is quantized, `False` otherwise.
`Tensor.is_meta`	Is `True` if the Tensor is a meta tensor, `False` otherwise.
`Tensor.device`	Is the `torch.device` where this Tensor is.

`Tensor.grad`	This attribute is `None` by default and becomes a Tensor the first time a call to `backward()` computes gradients for `self`.
`Tensor.ndim`	Alias for `dim()`
`Tensor.real`	Returns a new tensor containing real values of the `self` tensor for a complex-valued input tensor.
`Tensor.imag`	Returns a new tensor containing imaginary values of the `self` tensor.
`Tensor.nbytes`	Returns the number of bytes consumed by the "view" of elements of the Tensor if the Tensor does not use sparse storage layout.
`Tensor.itemsize`	Alias for `element_size()`
`Tensor.abs`	See `torch.abs()`
`Tensor.abs_`	In-place version of `abs()`
`Tensor.absolute`	Alias for `abs()`
`Tensor.absolute_`	In-place version of `absolute()` Alias for `abs_()`
`Tensor.acos`	See `torch.acos()`
`Tensor.acos_`	In-place version of `acos()`
`Tensor.arccos`	See `torch.arccos()`
`Tensor.arccos_`	In-place version of `arccos()`
`Tensor.add`	Add a scalar or tensor to `self` tensor.
`Tensor.add_`	In-place version of `add()`
`Tensor.addbmm`	See `torch.addbmm()`
`Tensor.addbmm_`	In-place version of `addbmm()`
`Tensor.addcdiv`	See `torch.addcdiv()`
`Tensor.addcdiv_`	In-place version of `addcdiv()`
`Tensor.addcmul`	See `torch.addcmul()`
`Tensor.addcmul_`	In-place version of `addcmul()`
`Tensor.addmm`	See `torch.addmm()`
`Tensor.addmm_`	In-place version of `addmm()`
`Tensor.sspaddmm`	See `torch.sspaddmm()`
`Tensor.addmv`	See `torch.addmv()`

`Tensor.addmv_`	In-place version of `addmv()`
`Tensor.addr`	See `torch.addr()`
`Tensor.addr_`	In-place version of `addr()`
`Tensor.adjoint`	Alias for `adjoint()`
`Tensor.allclose`	See `torch.allclose()`
`Tensor.amax`	See `torch.amax()`
`Tensor.amin`	See `torch.amin()`
`Tensor.aminmax`	See `torch.aminmax()`
`Tensor.angle`	See `torch.angle()`
`Tensor.apply_`	Applies the function `callable` to each element in the tensor, replacing each element with the value returned by `callable`.
`Tensor.argmax`	See `torch.argmax()`
`Tensor.argmin`	See `torch.argmin()`
`Tensor.argsort`	See `torch.argsort()`
`Tensor.argwhere`	See `torch.argwhere()`
`Tensor.asin`	See `torch.asin()`
`Tensor.asin_`	In-place version of `asin()`
`Tensor.arcsin`	See `torch.arcsin()`
`Tensor.arcsin_`	In-place version of `arcsin()`
`Tensor.as_strided`	See `torch.as_strided()`
`Tensor.atan`	See `torch.atan()`
`Tensor.atan_`	In-place version of `atan()`
`Tensor.arctan`	See `torch.arctan()`
`Tensor.arctan_`	In-place version of `arctan()`
`Tensor.atan2`	See `torch.atan2()`
`Tensor.atan2_`	In-place version of `atan2()`
`Tensor.arctan2`	See `torch.arctan2()`
`Tensor.arctan2_`	atan2_(other) -> Tensor

`Tensor.all`	See `torch.all()`
`Tensor.any`	See `torch.any()`
`Tensor.backward`	Computes the gradient of current tensor wrt graph leaves.
`Tensor.baddbmm`	See `torch.baddbmm()`
`Tensor.baddbmm_`	In-place version of `baddbmm()`
`Tensor.bernoulli`	Returns a result tensor where each `result[i]` is independently sampled from $\text{Bernoulli}(\texttt{self[i]})$.
`Tensor.bernoulli_`	Fills each location of `self` with an independent sample from $\text{Bernoulli}(\texttt{p})$.
`Tensor.bfloat16`	`self.bfloat16()` is equivalent to `self.to(torch.bfloat16)`.
`Tensor.bincount`	See `torch.bincount()`
`Tensor.bitwise_not`	See `torch.bitwise_not()`
`Tensor.bitwise_not_`	In-place version of `bitwise_not()`
`Tensor.bitwise_and`	See `torch.bitwise_and()`
`Tensor.bitwise_and_`	In-place version of `bitwise_and()`
`Tensor.bitwise_or`	See `torch.bitwise_or()`
`Tensor.bitwise_or_`	In-place version of `bitwise_or()`
`Tensor.bitwise_xor`	See `torch.bitwise_xor()`
`Tensor.bitwise_xor_`	In-place version of `bitwise_xor()`
`Tensor.bitwise_left_shift`	See `torch.bitwise_left_shift()`
`Tensor.bitwise_left_shift_`	In-place version of `bitwise_left_shift()`
`Tensor.bitwise_right_shift`	See `torch.bitwise_right_shift()`
`Tensor.bitwise_right_shift_`	In-place version of `bitwise_right_shift()`
`Tensor.bmm`	See `torch.bmm()`
`Tensor.bool`	`self.bool()` is equivalent to `self.to(torch.bool)`.
`Tensor.byte`	`self.byte()` is equivalent to `self.to(torch.uint8)`.
`Tensor.broadcast_to`	See `torch.broadcast_to()`.
`Tensor.cauchy_`	Fills the tensor with numbers drawn from the Cauchy distribution:
`Tensor.ceil`	See `torch.ceil()`

`Tensor.ceil_`	In-place version of `ceil()`
`Tensor.char`	`self.char()` is equivalent to `self.to(torch.int8)`.
`Tensor.cholesky`	See `torch.cholesky()`
`Tensor.cholesky_inverse`	See `torch.cholesky_inverse()`
`Tensor.cholesky_solve`	See `torch.cholesky_solve()`
`Tensor.chunk`	See `torch.chunk()`
`Tensor.clamp`	See `torch.clamp()`
`Tensor.clamp_`	In-place version of `clamp()`
`Tensor.clip`	Alias for `clamp()`.
`Tensor.clip_`	Alias for `clamp_()`.
`Tensor.clone`	See `torch.clone()`
`Tensor.contiguous`	Returns a contiguous in memory tensor containing the same data as `self` tensor.
`Tensor.copy_`	Copies the elements from `src` into `self` tensor and returns `self`.
`Tensor.conj`	See `torch.conj()`
`Tensor.conj_physical`	See `torch.conj_physical()`
`Tensor.conj_physical_`	In-place version of `conj_physical()`
`Tensor.resolve_conj`	See `torch.resolve_conj()`
`Tensor.resolve_neg`	See `torch.resolve_neg()`
`Tensor.copysign`	See `torch.copysign()`
`Tensor.copysign_`	In-place version of `copysign()`
`Tensor.cos`	See `torch.cos()`
`Tensor.cos_`	In-place version of `cos()`
`Tensor.cosh`	See `torch.cosh()`
`Tensor.cosh_`	In-place version of `cosh()`
`Tensor.corrcoef`	See `torch.corrcoef()`
`Tensor.count_nonzero`	See `torch.count_nonzero()`
`Tensor.cov`	See `torch.cov()`

`Tensor.acosh`	See `torch.acosh()`
`Tensor.acosh_`	In-place version of `acosh()`
`Tensor.arccosh`	acosh() -> Tensor
`Tensor.arccosh_`	acosh_() -> Tensor
`Tensor.cpu`	Returns a copy of this object in CPU memory.
`Tensor.cross`	See `torch.cross()`
`Tensor.cuda`	Returns a copy of this object in CUDA memory.
`Tensor.logcumsumexp`	See `torch.logcumsumexp()`
`Tensor.cummax`	See `torch.cummax()`
`Tensor.cummin`	See `torch.cummin()`
`Tensor.cumprod`	See `torch.cumprod()`
`Tensor.cumprod_`	In-place version of `cumprod()`
`Tensor.cumsum`	See `torch.cumsum()`
`Tensor.cumsum_`	In-place version of `cumsum()`
`Tensor.chalf`	`self.chalf()` is equivalent to `self.to(torch.complex32)`.
`Tensor.cfloat`	`self.cfloat()` is equivalent to `self.to(torch.complex64)`.
`Tensor.cdouble`	`self.cdouble()` is equivalent to `self.to(torch.complex128)`.
`Tensor.data_ptr`	Returns the address of the first element of `self` tensor.
`Tensor.deg2rad`	See `torch.deg2rad()`
`Tensor.dequantize`	Given a quantized Tensor, dequantize it and return the dequantized float Tensor.
`Tensor.det`	See `torch.det()`
`Tensor.dense_dim`	Return the number of dense dimensions in a sparse tensor `self`.
`Tensor.detach`	Returns a new Tensor, detached from the current graph.
`Tensor.detach_`	Detaches the Tensor from the graph that created it, making it a leaf.
`Tensor.diag`	See `torch.diag()`
`Tensor.diag_embed`	See `torch.diag_embed()`
`Tensor.diagflat`	See `torch.diagflat()`

`Tensor.diagonal`	See `torch.diagonal()`
`Tensor.diagonal_scatter`	See `torch.diagonal_scatter()`
`Tensor.fill_diagonal_`	Fill the main diagonal of a tensor that has at least 2-dimensions.
`Tensor.fmax`	See `torch.fmax()`
`Tensor.fmin`	See `torch.fmin()`
`Tensor.diff`	See `torch.diff()`
`Tensor.digamma`	See `torch.digamma()`
`Tensor.digamma_`	In-place version of `digamma()`
`Tensor.dim`	Returns the number of dimensions of `self` tensor.
`Tensor.dim_order`	Returns a tuple of int describing the dim order or physical layout of `self`.
`Tensor.dist`	See `torch.dist()`
`Tensor.div`	See `torch.div()`
`Tensor.div_`	In-place version of `div()`
`Tensor.divide`	See `torch.divide()`
`Tensor.divide_`	In-place version of `divide()`
`Tensor.dot`	See `torch.dot()`
`Tensor.double`	`self.double()` is equivalent to `self.to(torch.float64)`.
`Tensor.dsplit`	See `torch.dsplit()`
`Tensor.element_size`	Returns the size in bytes of an individual element.
`Tensor.eq`	See `torch.eq()`
`Tensor.eq_`	In-place version of `eq()`
`Tensor.equal`	See `torch.equal()`
`Tensor.erf`	See `torch.erf()`
`Tensor.erf_`	In-place version of `erf()`
`Tensor.erfc`	See `torch.erfc()`
`Tensor.erfc_`	In-place version of `erfc()`
`Tensor.erfinv`	See `torch.erfinv()`

`Tensor.erfinv_`	In-place version of `erfinv()`
`Tensor.exp`	See `torch.exp()`
`Tensor.exp_`	In-place version of `exp()`
`Tensor.expm1`	See `torch.expm1()`
`Tensor.expm1_`	In-place version of `expm1()`
`Tensor.expand`	Returns a new view of the `self` tensor with singleton dimensions expanded to a larger size.
`Tensor.expand_as`	Expand this tensor to the same size as `other`.
`Tensor.exponential_`	Fills `self` tensor with elements drawn from the exponential distribution:
`Tensor.fix`	See `torch.fix()`.
`Tensor.fix_`	In-place version of `fix()`
`Tensor.fill_`	Fills `self` tensor with the specified value.
`Tensor.flatten`	See `torch.flatten()`
`Tensor.flip`	See `torch.flip()`
`Tensor.fliplr`	See `torch.fliplr()`
`Tensor.flipud`	See `torch.flipud()`
`Tensor.float`	`self.float()` is equivalent to `self.to(torch.float32)`.
`Tensor.float_power`	See `torch.float_power()`
`Tensor.float_power_`	In-place version of `float_power()`
`Tensor.floor`	See `torch.floor()`
`Tensor.floor_`	In-place version of `floor()`
`Tensor.floor_divide`	See `torch.floor_divide()`
`Tensor.floor_divide_`	In-place version of `floor_divide()`
`Tensor.fmod`	See `torch.fmod()`
`Tensor.fmod_`	In-place version of `fmod()`
`Tensor.frac`	See `torch.frac()`
`Tensor.frac_`	In-place version of `frac()`
`Tensor.frexp`	See `torch.frexp()`

`Tensor.gather`	See `torch.gather()`
`Tensor.gcd`	See `torch.gcd()`
`Tensor.gcd_`	In-place version of `gcd()`
`Tensor.ge`	See `torch.ge()`.
`Tensor.ge_`	In-place version of `ge()`.
`Tensor.greater_equal`	See `torch.greater_equal()`.
`Tensor.greater_equal_`	In-place version of `greater_equal()`.
`Tensor.geometric_`	Fills `self` tensor with elements drawn from the geometric distribution:
`Tensor.geqrf`	See `torch.geqrf()`
`Tensor.ger`	See `torch.ger()`
`Tensor.get_device`	For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides.
`Tensor.gt`	See `torch.gt()`.
`Tensor.gt_`	In-place version of `gt()`.
`Tensor.greater`	See `torch.greater()`.
`Tensor.greater_`	In-place version of `greater()`.
`Tensor.half`	`self.half()` is equivalent to `self.to(torch.float16)`.
`Tensor.hardshrink`	See `torch.nn.functional.hardshrink()`
`Tensor.heaviside`	See `torch.heaviside()`
`Tensor.histc`	See `torch.histc()`
`Tensor.histogram`	See `torch.histogram()`
`Tensor.hsplit`	See `torch.hsplit()`
`Tensor.hypot`	See `torch.hypot()`
`Tensor.hypot_`	In-place version of `hypot()`
`Tensor.i0`	See `torch.i0()`
`Tensor.i0_`	In-place version of `i0()`
`Tensor.igamma`	See `torch.igamma()`
`Tensor.igamma_`	In-place version of `igamma()`

`Tensor.igammac`	See `torch.igammac()`
`Tensor.igammac_`	In-place version of `igammac()`
`Tensor.index_add_`	Accumulate the elements of `alpha` times `source` into the `self` tensor by adding to the indices in the order given in `index`.
`Tensor.index_add`	Out-of-place version of `torch.Tensor.index_add_()`.
`Tensor.index_copy_`	Copies the elements of `tensor` into the `self` tensor by selecting the indices in the order given in `index`.
`Tensor.index_copy`	Out-of-place version of `torch.Tensor.index_copy_()`.
`Tensor.index_fill_`	Fills the elements of the `self` tensor with value `value` by selecting the indices in the order given in `index`.
`Tensor.index_fill`	Out-of-place version of `torch.Tensor.index_fill_()`.
`Tensor.index_put_`	Puts values from the tensor `values` into the tensor `self` using the indices specified in `indices` (which is a tuple of Tensors).
`Tensor.index_put`	Out-place version of `index_put_()`.
`Tensor.index_reduce_`	Accumulate the elements of `source` into the `self` tensor by accumulating to the indices in the order given in `index` using the reduction given by the `reduce` argument.
`Tensor.index_reduce`	
`Tensor.index_select`	See `torch.index_select()`
`Tensor.indices`	Return the indices tensor of a sparse COO tensor.
`Tensor.inner`	See `torch.inner()`.
`Tensor.int`	`self.int()` is equivalent to `self.to(torch.int32)`.
`Tensor.int_repr`	Given a quantized Tensor, `self.int_repr()` returns a CPU Tensor with uint8_t as data type that stores the underlying uint8_t values of the given Tensor.
`Tensor.inverse`	See `torch.inverse()`
`Tensor.isclose`	See `torch.isclose()`
`Tensor.isfinite`	See `torch.isfinite()`
`Tensor.isinf`	See `torch.isinf()`
`Tensor.isposinf`	See `torch.isposinf()`
`Tensor.isneginf`	See `torch.isneginf()`
`Tensor.isnan`	See `torch.isnan()`
`Tensor.is_contiguous`	Returns True if `self` tensor is contiguous in memory in the order specified by memory format.

`Tensor.is_complex`	Returns True if the data type of `self` is a complex data type.
`Tensor.is_conj`	Returns True if the conjugate bit of `self` is set to true.
`Tensor.is_floating_point`	Returns True if the data type of `self` is a floating point data type.
`Tensor.is_inference`	See `torch.is_inference()`
`Tensor.is_leaf`	All Tensors that have `requires_grad` which is `False` will be leaf Tensors by convention.
`Tensor.is_pinned`	Returns true if this tensor resides in pinned memory.
`Tensor.is_set_to`	Returns True if both tensors are pointing to the exact same memory (same storage, offset, size and stride).
`Tensor.is_shared`	Checks if tensor is in shared memory.
`Tensor.is_signed`	Returns True if the data type of `self` is a signed data type.
`Tensor.is_sparse`	Is `True` if the Tensor uses sparse COO storage layout, `False` otherwise.
`Tensor.istft`	See `torch.istft()`
`Tensor.isreal`	See `torch.isreal()`
`Tensor.item`	Returns the value of this tensor as a standard Python number.
`Tensor.kthvalue`	See `torch.kthvalue()`
`Tensor.lcm`	See `torch.lcm()`
`Tensor.lcm_`	In-place version of `lcm()`
`Tensor.ldexp`	See `torch.ldexp()`
`Tensor.ldexp_`	In-place version of `ldexp()`
`Tensor.le`	See `torch.le()`.
`Tensor.le_`	In-place version of `le()`.
`Tensor.less_equal`	See `torch.less_equal()`.
`Tensor.less_equal_`	In-place version of `less_equal()`.
`Tensor.lerp`	See `torch.lerp()`
`Tensor.lerp_`	In-place version of `lerp()`
`Tensor.lgamma`	See `torch.lgamma()`
`Tensor.lgamma_`	In-place version of `lgamma()`
`Tensor.log`	See `torch.log()`

`Tensor.log_`	In-place version of `log()`
`Tensor.logdet`	See `torch.logdet()`
`Tensor.log10`	See `torch.log10()`
`Tensor.log10_`	In-place version of `log10()`
`Tensor.log1p`	See `torch.log1p()`
`Tensor.log1p_`	In-place version of `log1p()`
`Tensor.log2`	See `torch.log2()`
`Tensor.log2_`	In-place version of `log2()`
`Tensor.log_normal_`	Fills `self` tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ.
`Tensor.logaddexp`	See `torch.logaddexp()`
`Tensor.logaddexp2`	See `torch.logaddexp2()`
`Tensor.logsumexp`	See `torch.logsumexp()`
`Tensor.logical_and`	See `torch.logical_and()`
`Tensor.logical_and_`	In-place version of `logical_and()`
`Tensor.logical_not`	See `torch.logical_not()`
`Tensor.logical_not_`	In-place version of `logical_not()`
`Tensor.logical_or`	See `torch.logical_or()`
`Tensor.logical_or_`	In-place version of `logical_or()`
`Tensor.logical_xor`	See `torch.logical_xor()`
`Tensor.logical_xor_`	In-place version of `logical_xor()`
`Tensor.logit`	See `torch.logit()`
`Tensor.logit_`	In-place version of `logit()`
`Tensor.long`	`self.long()` is equivalent to `self.to(torch.int64)`.
`Tensor.lt`	See `torch.lt()`.
`Tensor.lt_`	In-place version of `lt()`.
`Tensor.less`	lt(other) -> Tensor
`Tensor.less_`	In-place version of `less()`.

`Tensor.lu`	See `torch.lu()`
`Tensor.lu_solve`	See `torch.lu_solve()`
`Tensor.as_subclass`	Makes a `cls` instance with the same data pointer as `self`.
`Tensor.map_`	Applies `callable` for each element in `self` tensor and the given `tensor` and stores the results in `self` tensor.
`Tensor.masked_scatter_`	Copies elements from `source` into `self` tensor at positions where the `mask` is True.
`Tensor.masked_scatter`	Out-of-place version of `torch.Tensor.masked_scatter_()`
`Tensor.masked_fill_`	Fills elements of `self` tensor with `value` where `mask` is True.
`Tensor.masked_fill`	Out-of-place version of `torch.Tensor.masked_fill_()`
`Tensor.masked_select`	See `torch.masked_select()`
`Tensor.matmul`	See `torch.matmul()`
`Tensor.matrix_power`	• NOTE `matrix_power()` is deprecated, use `torch.linalg.matrix_power()` instead.
`Tensor.matrix_exp`	See `torch.matrix_exp()`
`Tensor.max`	See `torch.max()`
`Tensor.maximum`	See `torch.maximum()`
`Tensor.mean`	See `torch.mean()`
`Tensor.nanmean`	See `torch.nanmean()`
`Tensor.median`	See `torch.median()`
`Tensor.nanmedian`	See `torch.nanmedian()`
`Tensor.min`	See `torch.min()`
`Tensor.minimum`	See `torch.minimum()`
`Tensor.mm`	See `torch.mm()`
`Tensor.smm`	See `torch.smm()`
`Tensor.mode`	See `torch.mode()`
`Tensor.movedim`	See `torch.movedim()`
`Tensor.moveaxis`	See `torch.moveaxis()`
`Tensor.msort`	See `torch.msort()`

`Tensor.mul`	See `torch.mul()`.
`Tensor.mul_`	In-place version of `mul()`.
`Tensor.multiply`	See `torch.multiply()`.
`Tensor.multiply_`	In-place version of `multiply()`.
`Tensor.multinomial`	See `torch.multinomial()`
`Tensor.mv`	See `torch.mv()`
`Tensor.mvlgamma`	See `torch.mvlgamma()`
`Tensor.mvlgamma_`	In-place version of `mvlgamma()`
`Tensor.nansum`	See `torch.nansum()`
`Tensor.narrow`	See `torch.narrow()`.
`Tensor.narrow_copy`	See `torch.narrow_copy()`.
`Tensor.ndimension`	Alias for `dim()`
`Tensor.nan_to_num`	See `torch.nan_to_num()`.
`Tensor.nan_to_num_`	In-place version of `nan_to_num()`.
`Tensor.ne`	See `torch.ne()`.
`Tensor.ne_`	In-place version of `ne()`.
`Tensor.not_equal`	See `torch.not_equal()`.
`Tensor.not_equal_`	In-place version of `not_equal()`.
`Tensor.neg`	See `torch.neg()`
`Tensor.neg_`	In-place version of `neg()`
`Tensor.negative`	See `torch.negative()`
`Tensor.negative_`	In-place version of `negative()`
`Tensor.nelement`	Alias for `numel()`
`Tensor.nextafter`	See `torch.nextafter()`
`Tensor.nextafter_`	In-place version of `nextafter()`
`Tensor.nonzero`	See `torch.nonzero()`
`Tensor.norm`	See `torch.norm()`

`Tensor.normal_`	Fills `self` tensor with elements samples from the normal distribution parameterized by `mean` and `std`.
`Tensor.numel`	See `torch.numel()`
`Tensor.numpy`	Returns the tensor as a NumPy `ndarray`.
`Tensor.orgqr`	See `torch.orgqr()`
`Tensor.ormqr`	See `torch.ormqr()`
`Tensor.outer`	See `torch.outer()`.
`Tensor.permute`	See `torch.permute()`
`Tensor.pin_memory`	Copies the tensor to pinned memory, if it's not already pinned.
`Tensor.pinverse`	See `torch.pinverse()`
`Tensor.polygamma`	See `torch.polygamma()`
`Tensor.polygamma_`	In-place version of `polygamma()`
`Tensor.positive`	See `torch.positive()`
`Tensor.pow`	See `torch.pow()`
`Tensor.pow_`	In-place version of `pow()`
`Tensor.prod`	See `torch.prod()`
`Tensor.put_`	Copies the elements from `source` into the positions specified by `index`.
`Tensor.qr`	See `torch.qr()`
`Tensor.qscheme`	Returns the quantization scheme of a given QTensor.
`Tensor.quantile`	See `torch.quantile()`
`Tensor.nanquantile`	See `torch.nanquantile()`
`Tensor.q_scale`	Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().
`Tensor.q_zero_point`	Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().
`Tensor.q_per_channel_scales`	Given a Tensor quantized by linear (affine) per-channel quantization, returns a Tensor of scales of the underlying quantizer.
`Tensor.q_per_channel_zero_points`	Given a Tensor quantized by linear (affine) per-channel quantization, returns a tensor of zero_points of the underlying quantizer.
`Tensor.q_per_channel_axis`	Given a Tensor quantized by linear (affine) per-channel quantization, returns the index of dimension on which per-channel quantization is applied.

`Tensor.rad2deg`	See `torch.rad2deg()`
`Tensor.random_`	Fills `self` tensor with numbers sampled from the discrete uniform distribution over `[from, to - 1]`.
`Tensor.ravel`	see `torch.ravel()`
`Tensor.reciprocal`	See `torch.reciprocal()`
`Tensor.reciprocal_`	In-place version of `reciprocal()`
`Tensor.record_stream`	Ensures that the tensor memory is not reused for another tensor until all current work queued on `stream` are complete.
`Tensor.register_hook`	Registers a backward hook.
`Tensor.register_post_accumulate_grad_hook`	Registers a backward hook that runs after grad accumulation.
`Tensor.remainder`	See `torch.remainder()`
`Tensor.remainder_`	In-place version of `remainder()`
`Tensor.renorm`	See `torch.renorm()`
`Tensor.renorm_`	In-place version of `renorm()`
`Tensor.repeat`	Repeats this tensor along the specified dimensions.
`Tensor.repeat_interleave`	See `torch.repeat_interleave()`.
`Tensor.requires_grad`	Is `True` if gradients need to be computed for this Tensor, `False` otherwise.
`Tensor.requires_grad_`	Change if autograd should record operations on this tensor: sets this tensor's `requires_grad` attribute in-place.
`Tensor.reshape`	Returns a tensor with the same data and number of elements as `self` but with the specified shape.
`Tensor.reshape_as`	Returns this tensor as the same shape as `other`.
`Tensor.resize_`	Resizes `self` tensor to the specified size.
`Tensor.resize_as_`	Resizes the `self` tensor to be the same size as the specified `tensor`.
`Tensor.retain_grad`	Enables this Tensor to have their `grad` populated during `backward()`.
`Tensor.retains_grad`	Is `True` if this Tensor is non-leaf and its `grad` is enabled to be populated during `backward()`, `False` otherwise.
`Tensor.roll`	See `torch.roll()`
`Tensor.rot90`	See `torch.rot90()`
`Tensor.round`	See `torch.round()`
`Tensor.round_`	In-place version of `round()`

`Tensor.rsqrt`	See `torch.rsqrt()`
`Tensor.rsqrt_`	In-place version of `rsqrt()`
`Tensor.scatter`	Out-of-place version of `torch.Tensor.scatter_()`
`Tensor.scatter_`	Writes all values from the tensor `src` into `self` at the indices specified in the `index` tensor.
`Tensor.scatter_add_`	Adds all values from the tensor `src` into `self` at the indices specified in the `index` tensor in a similar fashion as `scatter_()`.
`Tensor.scatter_add`	Out-of-place version of `torch.Tensor.scatter_add_()`
`Tensor.scatter_reduce_`	Reduces all values from the `src` tensor to the indices specified in the `index` tensor in the `self` tensor using the applied reduction defined via the `reduce` argument (`"sum"`, `"prod"`, `"mean"`, `"amax"`, `"amin"`).
`Tensor.scatter_reduce`	Out-of-place version of `torch.Tensor.scatter_reduce_()`
`Tensor.select`	See `torch.select()`
`Tensor.select_scatter`	See `torch.select_scatter()`
`Tensor.set_`	Sets the underlying storage, size, and strides.
`Tensor.share_memory_`	Moves the underlying storage to shared memory.
`Tensor.short`	`self.short()` is equivalent to `self.to(torch.int16)`.
`Tensor.sigmoid`	See `torch.sigmoid()`
`Tensor.sigmoid_`	In-place version of `sigmoid()`
`Tensor.sign`	See `torch.sign()`
`Tensor.sign_`	In-place version of `sign()`
`Tensor.signbit`	See `torch.signbit()`
`Tensor.sgn`	See `torch.sgn()`
`Tensor.sgn_`	In-place version of `sgn()`
`Tensor.sin`	See `torch.sin()`
`Tensor.sin_`	In-place version of `sin()`
`Tensor.sinc`	See `torch.sinc()`
`Tensor.sinc_`	In-place version of `sinc()`
`Tensor.sinh`	See `torch.sinh()`
`Tensor.sinh_`	In-place version of `sinh()`

`Tensor.asinh`	See `torch.asinh()`
`Tensor.asinh_`	In-place version of `asinh()`
`Tensor.arcsinh`	See `torch.arcsinh()`
`Tensor.arcsinh_`	In-place version of `arcsinh()`
`Tensor.shape`	Returns the size of the `self` tensor.
`Tensor.size`	Returns the size of the `self` tensor.
`Tensor.slogdet`	See `torch.slogdet()`
`Tensor.slice_scatter`	See `torch.slice_scatter()`
`Tensor.softmax`	Alias for `torch.nn.functional.softmax()`.
`Tensor.sort`	See `torch.sort()`
`Tensor.split`	See `torch.split()`
`Tensor.sparse_mask`	Returns a new sparse tensor with values from a strided tensor `self` filtered by the indices of the sparse tensor `mask`.
`Tensor.sparse_dim`	Return the number of sparse dimensions in a sparse tensor `self`.
`Tensor.sqrt`	See `torch.sqrt()`
`Tensor.sqrt_`	In-place version of `sqrt()`
`Tensor.square`	See `torch.square()`
`Tensor.square_`	In-place version of `square()`
`Tensor.squeeze`	See `torch.squeeze()`
`Tensor.squeeze_`	In-place version of `squeeze()`
`Tensor.std`	See `torch.std()`
`Tensor.stft`	See `torch.stft()`
`Tensor.storage`	Returns the underlying `TypedStorage`.
`Tensor.untyped_storage`	Returns the underlying `UntypedStorage`.
`Tensor.storage_offset`	Returns `self` tensor's offset in the underlying storage in terms of number of storage elements (not bytes).
`Tensor.storage_type`	Returns the type of the underlying storage.
`Tensor.stride`	Returns the stride of `self` tensor.
`Tensor.sub`	See `torch.sub()`.

`Tensor.sub_`	In-place version of `sub()`
`Tensor.subtract`	See `torch.subtract()`.
`Tensor.subtract_`	In-place version of `subtract()`.
`Tensor.sum`	See `torch.sum()`
`Tensor.sum_to_size`	Sum `this` tensor to `size`.
`Tensor.svd`	See `torch.svd()`
`Tensor.swapaxes`	See `torch.swapaxes()`
`Tensor.swapdims`	See `torch.swapdims()`
`Tensor.t`	See `torch.t()`
`Tensor.t_`	In-place version of `t()`
`Tensor.tensor_split`	See `torch.tensor_split()`
`Tensor.tile`	See `torch.tile()`
`Tensor.to`	Performs Tensor dtype and/or device conversion.
`Tensor.to_mkldnn`	Returns a copy of the tensor in `torch.mkldnn` layout.
`Tensor.take`	See `torch.take()`
`Tensor.take_along_dim`	See `torch.take_along_dim()`
`Tensor.tan`	See `torch.tan()`
`Tensor.tan_`	In-place version of `tan()`
`Tensor.tanh`	See `torch.tanh()`
`Tensor.tanh_`	In-place version of `tanh()`
`Tensor.atanh`	See `torch.atanh()`
`Tensor.atanh_`	In-place version of `atanh()`
`Tensor.arctanh`	See `torch.arctanh()`
`Tensor.arctanh_`	In-place version of `arctanh()`
`Tensor.tolist`	Returns the tensor as a (nested) list.
`Tensor.topk`	See `torch.topk()`
`Tensor.to_dense`	Creates a strided copy of `self` if `self` is not a strided tensor, otherwise returns `self`.

`Tensor.to_sparse`	Returns a sparse copy of the tensor.
`Tensor.to_sparse_csr`	Convert a tensor to compressed row storage format (CSR).
`Tensor.to_sparse_csc`	Convert a tensor to compressed column storage (CSC) format.
`Tensor.to_sparse_bsr`	Convert a tensor to a block sparse row (BSR) storage format of given blocksize.
`Tensor.to_sparse_bsc`	Convert a tensor to a block sparse column (BSC) storage format of given blocksize.
`Tensor.trace`	See `torch.trace()`
`Tensor.transpose`	See `torch.transpose()`
`Tensor.transpose_`	In-place version of `transpose()`
`Tensor.triangular_solve`	See `torch.triangular_solve()`
`Tensor.tril`	See `torch.tril()`
`Tensor.tril_`	In-place version of `tril()`
`Tensor.triu`	See `torch.triu()`
`Tensor.triu_`	In-place version of `triu()`
`Tensor.true_divide`	See `torch.true_divide()`
`Tensor.true_divide_`	In-place version of `true_divide_()`
`Tensor.trunc`	See `torch.trunc()`
`Tensor.trunc_`	In-place version of `trunc()`
`Tensor.type`	Returns the type if *dtype* is not provided, else casts this object to the specified type.
`Tensor.type_as`	Returns this tensor cast to the type of the given tensor.
`Tensor.unbind`	See `torch.unbind()`
`Tensor.unflatten`	See `torch.unflatten()`.
`Tensor.unfold`	Returns a view of the original tensor which contains all slices of size `size` from `self` tensor in the dimension `dimension`.
`Tensor.uniform_`	Fills `self` tensor with numbers sampled from the continuous uniform distribution:
`Tensor.unique`	Returns the unique elements of the input tensor.
`Tensor.unique_consecutive`	Eliminates all but the first element from every consecutive group of equivalent elements.
`Tensor.unsqueeze`	See `torch.unsqueeze()`
`Tensor.unsqueeze_`	In-place version of `unsqueeze()`

`Tensor.values`	Return the values tensor of a sparse COO tensor.
`Tensor.var`	See `torch.var()`
`Tensor.vdot`	See `torch.vdot()`
`Tensor.view`	Returns a new tensor with the same data as the `self` tensor but of a different `shape` .

TORCH.CAT

torch.cat(*tensors*, *dim=0*, ***, *out=None*) → Tensor

Concatenates the given sequence of `seq` tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

`torch.cat()` can be seen as an inverse operation for `torch.split()` and `torch.chunk()`.

`torch.cat()` can be best understood via examples.

SEE ALSO

`torch.stack()` concatenates the given sequence along a new dimension.

Parameters

- **tensors** (*sequence of Tensors*) – any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.
- **dim** (*int, optional*) – the dimension over which the tensors are concatenated

Keyword Arguments

out (*Tensor, optional*) – the output tensor.

Example:

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

TORCH.MANUAL_SEED

torch.manual_seed(*seed*) [SOURCE]

Sets the seed for generating random numbers. Returns a *torch.Generator* object.

Parameters

seed (*int*) – The desired seed. Value must be within the inclusive range [*-0x8000_0000_0000_0000, 0xffff_ffff_ffff_ffff*]. Otherwise, a RuntimeError is raised. Negative inputs are remapped to positive values with the formula *0xffff_ffff_ffff_ffff + seed*.

Return type

Generator

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >

TORCH.MAX

torch.max(*input*) → Tensor

Returns the maximum value of all elements in the `input` tensor.

• WARNING

This function produces deterministic (sub)gradients unlike `max(dim=0)`

Parameters

input (*Tensor*) – the input tensor.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.6763,  0.7445, -2.2369]])
>>> torch.max(a)
tensor(0.7445)
```

torch.max(*input*, *dim*, *keepdim=False*, ***, *out=None*)

Returns a namedtuple `(values, indices)` where `values` is the maximum value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each maximum value found (argmax).

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see `torch.squeeze()`), resulting in the output tensors having 1 fewer dimension than `input`.

• NOTE

If there are multiple maximal values in a reduced row then the indices of the first maximal value are returned.

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to reduce.
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not. Default: `False`.

Keyword Arguments

out (*tuple*, *optional*) – the result tuple of two output tensors (max, max_indices)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-1.2360, -0.2942, -0.1222,  0.8475],
        [ 1.1949, -1.1127, -2.2379, -0.6702],
        [ 1.5717, -0.9207,  0.1297, -1.8768],
        [-0.6172,  1.0036, -0.6060, -0.2432]])
>>> torch.max(a, 1)
torch.return_types.max(values=tensor([0.8475, 1.1949, 1.5717, 1.0036]), indices=tensor([3, 0, 0, 1]))
```

torch.max(*input*, *other*, ***, *out=None*) → Tensor

See `torch.maximum()`.

TORCH.MIN

torch.min(*input*) → Tensor

Returns the minimum value of all elements in the `input` tensor.

• WARNING

This function produces deterministic (sub)gradients unlike `min(dim=0)`

Parameters

input (*Tensor*) – the input tensor.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.6750,  1.0857,  1.7197]])
>>> torch.min(a)
tensor(0.6750)
```

torch.min(*input*, *dim*, *keepdim=False*, ***, *out=None*)

Returns a namedtuple `(values, indices)` where `values` is the minimum value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each minimum value found (argmin).

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see `torch.squeeze()`), resulting in the output tensors having 1 fewer dimension than `input`.

• NOTE

If there are multiple minimal values in a reduced row then the indices of the first minimal value are returned.

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to reduce.
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not.

Keyword Arguments

out (*tuple*, *optional*) – the tuple of two output tensors (min, min_indices)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.6248,  1.1334, -1.1899, -0.2803],
        [-1.4644, -0.2635, -0.3651,  0.6134],
        [ 0.2457,  0.0384,  1.0128,  0.7015],
        [-0.1153,  2.9849,  2.1458,  0.5788]])
>>> torch.min(a, 1)
torch.return_types.min(values=tensor([-1.1899, -1.4644,  0.0384, -0.1153]), indices=tensor([2, 0, 1, 0]))
```

torch.min(*input*, *other*, ***, *out=None*) → Tensor

See `torch.minimum()`.

Previous

Next

LINEAR

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Examples:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >

MSELOSS

CLASS torch.nn.MSELoss(*size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{‘mean’;} \\ \text{sum}(L), & \text{if reduction} = \text{‘sum’.} \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The mean operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

Parameters

- **size_average** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when `reduce` is `False`. Default: `True`
- **reduce** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*str*, *optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Target: $(*)$, same shape as the input.

Examples:

```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >

MODULE

CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

• NOTE

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

add_module(*name*, *module*) [SOURCE]

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn*) [SOURCE]

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch.nn.init).

Parameters

fn (`Module` -> None) – function to be applied to each submodule

Returns

self

Return type

Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() [SOURCE]

Casts all floating point parameters and buffers to `bfloat16` datatype.

• NOTE

This method modifies the module in-place.

Returns

self

Return type

Module

buffers(*recurse=True*) [SOURCE]

Returns an iterator over module buffers.

Parameters

recurse (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields

torch.Tensor – module buffer

Return type

Iterator[*Tensor*]

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() [SOURCE]

Returns an iterator over immediate children modules.

Yields

Module – a child module

Return type

Iterator[*Module*]

compile(**args*, ***kwargs*) [SOURCE]

Compile this Module's forward using `torch.compile()`.

This Module's *__call__* method is compiled and all arguments are passed as-is to `torch.compile()`.

See `torch.compile()` for details on the arguments for this function.

cpu() [SOURCE]

Moves all model parameters and buffers to the CPU.

• NOTE

This method modifies the module in-place.

Returns

self

Return type

Module

cuda(*device=None*) [SOURCE]

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

• NOTE

This method modifies the module in-place.

Parameters

device (*int*, *optional*) – if specified, all parameters will be copied to that device

Returns

self

Return type

Module

double() [SOURCE]

Casts all floating point parameters and buffers to `double` datatype.

• NOTE

This method modifies the module in-place.

Returns

self

Return type

Module

eval() [SOURCE]

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See Locally disabling gradient computation for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns

self

Return type

Module

extra_repr() [SOURCE]

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type

str

float() [SOURCE]

Casts all floating point parameters and buffers to `float` datatype.

• NOTE

This method modifies the module in-place.

Returns

self

Return type

Module

forward(**input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

• NOTE

Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target*) [SOURCE]

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters

target (*str*) – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns

The buffer referenced by `target`

Return type

torch.Tensor

Raises

AttributeError – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() [SOURCE]

Returns any extra state to include in the module's state_dict. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's *state_dict*().

Note that extra state should be picklable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns

Any extra state to store in the module's state_dict

Return type

object

get_parameter(*target*) [SOURCE]

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters

target (*str*) – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns

The Parameter referenced by `target`

Return type

torch.nn.Parameter

Raises

AttributeError – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target*) [SOURCE]

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters

target (*str*) – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns

The submodule referenced by `target`

Return type

torch.nn.Module

Raises

AttributeError – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() [SOURCE]

Casts all floating point parameters and buffers to `half` datatype.

• NOTE

This method modifies the module in-place.

Returns

self

Return type

Module

ipu(*device=None*) [SOURCE]

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

• NOTE

This method modifies the module in-place.

Parameters

device (*int*, *optional*) – if specified, all parameters will be copied to that device

Returns

self

Return type

Module

load_state_dict(*state_dict*, *strict=True*, *assign=False*) [SOURCE]

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

• WARNING

If `assign` is `True` the optimizer must be created after the call to `load_state_dict`.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`
- **assign** (*bool*, *optional*) – whether to assign items in the state dictionary to their corresponding keys in the module instead of copying them inplace into the module's current parameters and buffers. When `False`, the properties of the tensors in the current module are preserved while when `True`, the properties of the Tensors in the state dict are preserved. Default: `False`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type

`NamedTuple` with `missing_keys` and `unexpected_keys` fields

• NOTE

If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

modules() [SOURCE]

Returns an iterator over all modules in the network.

Yields

Module – a module in the network

Return type

Iterator[*Module*]

• NOTE

Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix=''*, *recurse=True*, *remove_duplicate=True*) [SOURCE]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*, *optional*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module. Defaults to True.
- **remove_duplicate** (*bool*, *optional*) – whether to remove the duplicated buffers in the result. Defaults to True.

Yields

(*str, torch.Tensor*) – Tuple containing the name and buffer

Return type

Iterator[*Tuple*[str, *Tensor*]]

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() [SOURCE]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields

(*str, Module*) – Tuple containing a name and child module

Return type

Iterator[*Tuple*[str, *Module*]]

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*) [SOURCE]

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** (*Optional*[*Set*[*Module*]]) – a memo to store the set of modules already added to the result
- **prefix** (*str*) – a prefix that will be added to the name of the module
- **remove_duplicate** (*bool*) – whether to remove the duplicated module instances in the result or not

Yields

(*str, Module*) – Tuple of name and module

• NOTE

Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*, *remove_duplicate=True*) [SOURCE]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.
- **remove_duplicate** (*bool, optional*) – whether to remove the duplicated parameters in the result. Defaults to True.

Yields

(*str, Parameter*) – Tuple containing the name and parameter

Return type

Iterator[*Tuple*[str, *Parameter*]]

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters(*recurse=True*) [SOURCE]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters

recurse (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields

Parameter – module parameter

Return type

Iterator[*Parameter*]

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook*) [SOURCE]

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_buffer(*name*, *tensor*, *persistent=True*) [SOURCE]

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*, ***, *prepend=False*, *with_kwargs=False*, *always_call=False*) [SOURCE]

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output.

If `with_kwargs` is `False` or not specified, the input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called. The hook should have the following signature:

```
hook(module, args, output) -> None or modified output
```

If `with_kwargs` is `True`, the forward hook will be passed the `kwargs` given to the forward function and be expected to return the output possibly modified. The hook should have the following signature:

```
hook(module, args, kwargs, output) -> None or modified output
```

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If `True`, the provided `hook` will be fired before all existing `forward` hooks on this `torch.nn.modules.Module`. Otherwise, the provided `hook` will be fired after all existing `forward` hooks on this `torch.nn.modules.Module`. Note that global `forward` hooks registered with `register_module_forward_hook()` will fire before all hooks registered by this method. Default: `False`
- **with_kwargs** (*bool*) – If `True`, the `hook` will be passed the kwargs given to the forward function. Default: `False`
- **always_call** (*bool*) – If `True` the `hook` will be run regardless of whether an exception is raised while calling the Module. Default: `False`

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*, ***, *prepend=False*, *with_kwargs=False*) [SOURCE]

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked.

If `with_kwargs` is false or not specified, the input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple). The hook should have the following signature:

```
hook(module, args) -> None or modified input
```

If `with_kwargs` is true, the forward pre-hook will be passed the kwargs given to the forward function. And if the hook modifies the input, both the args and kwargs should be returned. The hook should have the following signature:

```
hook(module, args, kwargs) -> None or a tuple of modified input and kwargs
```

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If true, the provided `hook` will be fired before all existing `forward_pre` hooks on this `torch.nn.modules.Module`. Otherwise, the provided `hook` will be fired after all existing `forward_pre` hooks on this `torch.nn.modules.Module`. Note that global `forward_pre` hooks registered with `register_module_forward_pre_hook()` will fire before all hooks registered by this method. Default: `False`
- **with_kwargs** (*bool*) – If true, the `hook` will be passed the kwargs given to the forward function. Default: `False`

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook*, *prepend=False*) [SOURCE]

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

• WARNING

Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Parameters

- **hook** (*Callable*) – The user-defined hook to be registered.
- **prepend** (*bool*) – If true, the provided `hook` will be fired before all existing `backward` hooks on this `torch.nn.modules.Module`. Otherwise, the provided `hook` will be fired after all existing `backward` hooks on this `torch.nn.modules.Module`. Note that global `backward` hooks registered with `register_module_full_backward_hook()` will fire before all hooks registered by this method.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_full_backward_pre_hook(*hook*, *prepend=False*) [SOURCE]

Registers a backward pre-hook on the module.

The hook will be called every time the gradients for the module are computed. The hook should have the following signature:

```
hook(module, grad_output) -> tuple[Tensor] or None
```

The `grad_output` is a tuple. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the output that will be used in place of `grad_output` in subsequent computations. Entries in `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

• WARNING

Modifying inputs inplace is not allowed when using backward hooks and will raise an error.

Parameters

- **hook** (*Callable*) – The user-defined hook to be registered.
- **prepend** (*bool*) – If true, the provided `hook` will be fired before all existing `backward_pre` hooks on this `torch.nn.modules.Module`. Otherwise, the provided `hook` will be fired after all existing `backward_pre` hooks on this `torch.nn.modules.Module`. Note that global `backward_pre` hooks registered with `register_module_full_backward_pre_hook()` will fire before all hooks registered by this method.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*) [SOURCE]

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature::

hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_module(*name*, *module*) [SOURCE]

Alias for `add_module()`.

register_parameter(*name*, *param*) [SOURCE]

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

register_state_dict_pre_hook(*hook*) [SOURCE]

These hooks will be called with arguments: `self`, `prefix`, and `keep_vars` before calling `state_dict` on `self`. The registered hooks can be used to perform pre-processing before the `state_dict` call is made.

requires_grad_(*requires_grad=True*) [SOURCE]

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See Locally disabling gradient computation for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters

requires_grad (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns

self

Return type

Module

set_extra_state(*state*) [SOURCE]

This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Parameters

state (*dict*) – Extra state from the *state_dict*

share_memory() [SOURCE]

See `torch.Tensor.share_memory_()`

Return type

T

state_dict(*, *destination: T_destination*, *prefix: str = ''*, *keep_vars: bool = False*) → T_destination [SOURCE]

state_dict(*, *prefix: str = ''*, *keep_vars: bool = False*) → Dict[str, Any]

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

• NOTE

The returned object is a shallow copy. It contains references to the module's parameters and buffers.

• WARNING

Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

• WARNING

Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns

a dictionary containing a whole state of the module

Return type

dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(*device: Optional[Union[int, device]] = ...*, *dtype: Optional[Union[dtype, str]] = ...*, *non_blocking: bool = ...*) → T [SOURCE]

to(*dtype: Union[dtype, str]*, *non_blocking: bool = ...*) → T

to(*tensor: Tensor*, *non_blocking: bool = ...*) → T

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*) [SOURCE]

to(*dtype*, *non_blocking=False*) [SOURCE]

to(*tensor*, *non_blocking=False*) [SOURCE]

to(*memory_format=torch.channels_last*) [SOURCE]

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype` s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

> • NOTE
>
> This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns

self

Return type

Module

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*, *recurse=True*) [SOURCE]

Moves the parameters and buffers to the specified device without copying storage.

Parameters

- **device** (`torch.device`) – The desired device of the parameters and buffers in this module.
- **recurse** (*bool*) – Whether parameters and buffers of submodules should be recursively moved to the specified device.

Returns

self

Return type

Module

train(*mode=True*) [SOURCE]

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout` , `BatchNorm` , etc.

Parameters

mode (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True` .

Returns

self

Return type

Module

type(*dst_type*) [SOURCE]

Casts all parameters and buffers to `dst_type` .

• NOTE

This method modifies the module in-place.

Parameters

dst_type (*type or string*) – the desired type

Returns

self

Return type

Module

xpu(*device=None*) [SOURCE]

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

• NOTE

This method modifies the module in-place.

Parameters

device (*int, optional*) – if specified, all parameters will be copied to that device

Returns

self

Return type

Module

zero_grad(*set_to_none=True*) [SOURCE]

Resets gradients of all model parameters. See similar function under `torch.optim.Optimizer` for more context.

Parameters

set_to_none (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

‹ Previous Next ›

RELU

CLASS torch.nn.ReLU(*inplace=False*) [SOURCE]

Applies the rectified linear unit function element-wise:

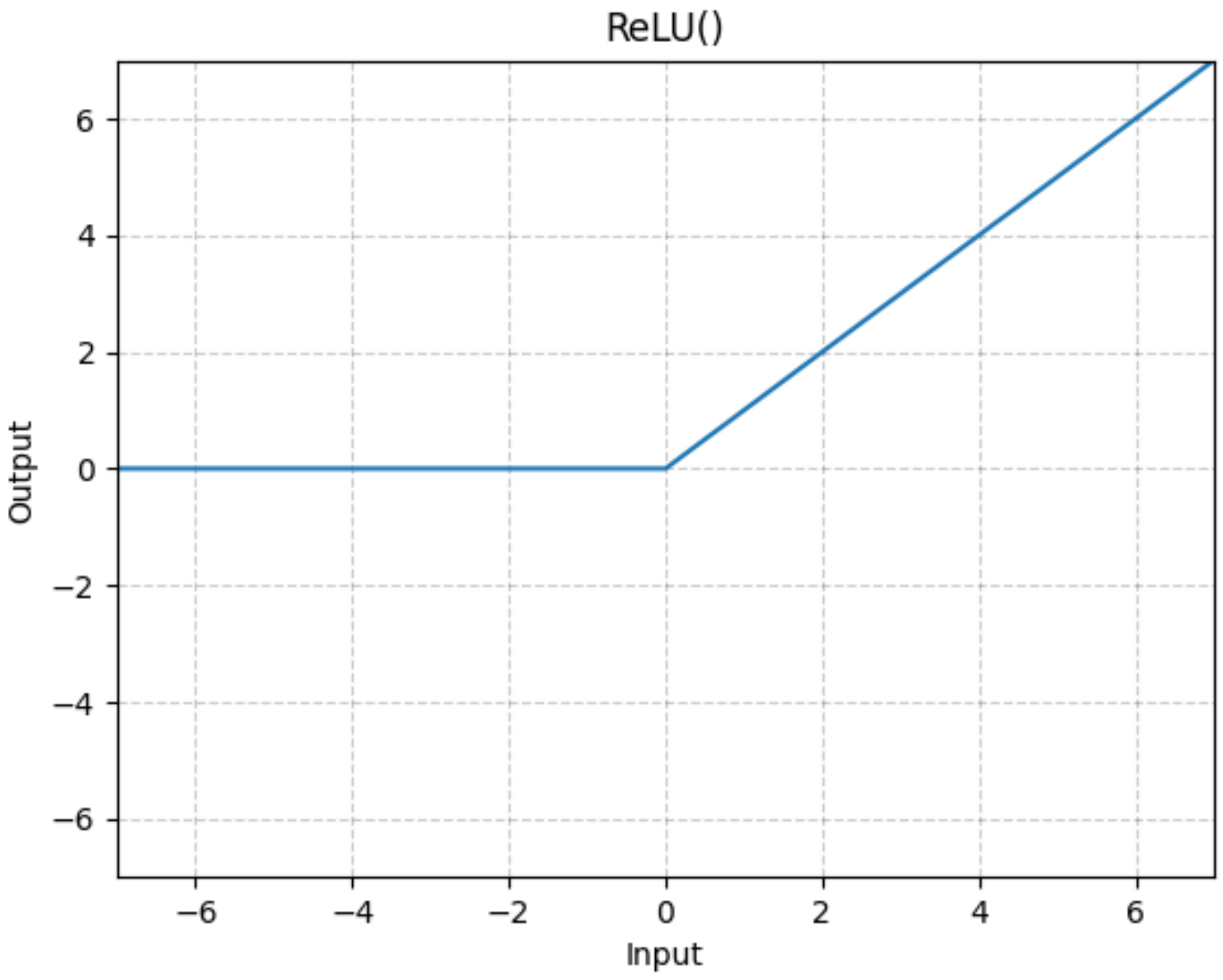
$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

Parameters

inplace (*bool*) – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Output: $(*)$, same shape as the input.



Examples:

```
>>> m = nn.ReLU()
>>> input = torch.randn(2)
>>> output = m(input)


An implementation of CReLU - https://arxiv.org/abs/1603.05201

>>> m = nn.ReLU()
>>> input = torch.randn(2).unsqueeze(0)
>>> output = torch.cat((m(input), m(-input)))
```

SIGMOID

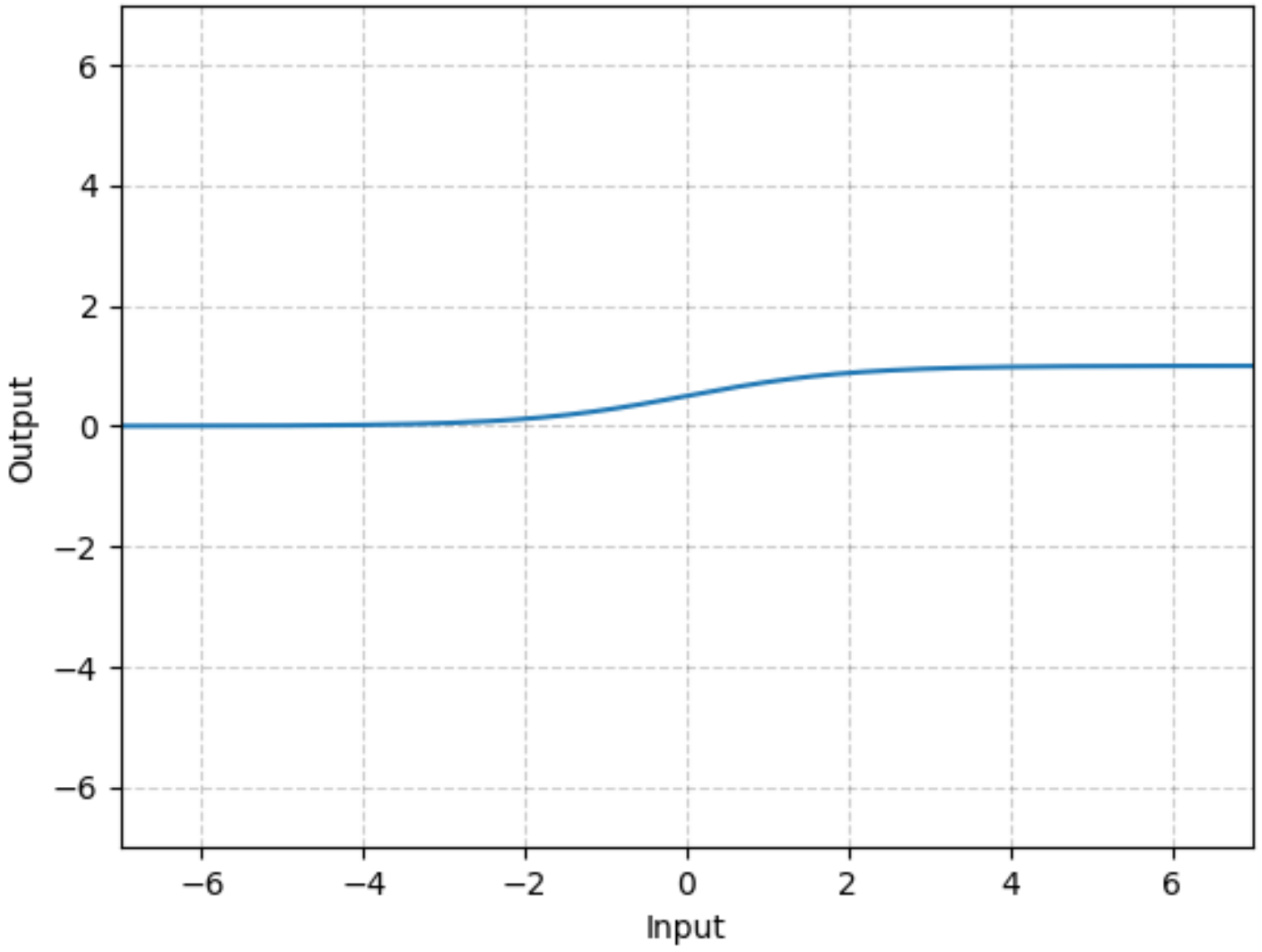
CLASS torch.nn.Sigmoid(*args, **kwargs) [SOURCE]

Applies the element-wise function:

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Output: $(*)$, same shape as the input.



Examples:

```
>>> m = nn.Sigmoid()
>>> input = torch.randn(2)
>>> output = m(input)
```

Previous

Next

SOFTMAX

CLASS torch.nn.Softmax(*dim=None*) [SOURCE]

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

When the input Tensor is a sparse tensor then the unspecified values are treated as `-inf`.

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Returns

a Tensor of the same dimension and shape as the input with values in the range [0, 1]

Parameters

dim (*int*) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

Return type

None

• NOTE

This module doesn't work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use *LogSoftmax* instead (it's faster and has better numerical properties).

Examples:

```
>>> m = nn.Softmax(dim=1)
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

Previous

Next

NO_GRAD

CLASS torch.no_grad(*orig_func=None*) [SOURCE]

Context-manager that disables gradient calculation.

Disabling gradient calculation is useful for inference, when you are sure that you will not call `Tensor.backward()`. It will reduce memory consumption for computations that would otherwise have *requires_grad=True*.

In this mode, the result of every computation will have *requires_grad=False*, even when the inputs have *requires_grad=True*. There is an exception! All factory functions, or functions that create a new Tensor and take a requires_grad kwarg, will NOT be affected by this mode.

This context manager is thread local; it will not affect computation in other threads.

Also functions as a decorator.

• NOTE

No-grad is one of several mechanisms that can enable or disable gradients locally see Locally disabling gradient computation for more information on how they compare.

• NOTE

This API does not apply to forward-mode AD. If you want to disable forward AD for a computation, you can unpack your dual tensors.

Example::

```
>>> x = torch.tensor([1.], requires_grad=True)
>>> with torch.no_grad():
...     y = x * 2
>>> y.requires_grad
False
>>> @torch.no_grad()
... def doubler(x):
...     return x * 2
>>> z = doubler(x)
>>> z.requires_grad
False
>>> @torch.no_grad
... def tripler(x):
...     return x * 3
>>> z = tripler(x)
>>> z.requires_grad
False
>>> # factory function exception
>>> with torch.no_grad():
...     a = torch.nn.Parameter(torch.rand(10))
>>> a.requires_grad
True
```

SGD

CLASS torch.optim.SGD(*params*, *lr=<required parameter>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, ***, *maximize=False*, *foreach=None*, *differentiable=False*) [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

$$
\begin{aligned}
&\rule{110mm}{0.4pt} \\
&\textbf{input} : \gamma \text{ (lr)}, \ \theta_0 \text{ (params)}, \ f(\theta) \text{ (objective)}, \ \lambda \text{ (weight decay)}, \\
&\hspace{13mm} \mu \text{ (momentum)}, \ \tau \text{ (dampening)}, \ \textit{nesterov}, \ \textit{maximize} \\
&\rule{110mm}{0.4pt} \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\hspace{5mm} g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\hspace{5mm} \textbf{if } \lambda \neq 0 \\
&\hspace{10mm} g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\hspace{5mm} \textbf{if } \mu \neq 0 \\
&\hspace{10mm} \textbf{if } t > 1 \\
&\hspace{15mm} \mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + (1 - \tau) g_t \\
&\hspace{10mm} \textbf{else} \\
&\hspace{15mm} \mathbf{b}_t \leftarrow g_t \\
&\hspace{10mm} \textbf{if } \textit{nesterov} \\
&\hspace{15mm} g_t \leftarrow g_t + \mu \mathbf{b}_t \\
&\hspace{10mm} \textbf{else} \\
&\hspace{15mm} g_t \leftarrow \mathbf{b}_t \\
&\hspace{5mm} \textbf{if } \textit{maximize} \\
&\hspace{10mm} \theta_t \leftarrow \theta_{t-1} + \gamma g_t \\
&\hspace{5mm} \textbf{else} \\
&\hspace{10mm} \theta_t \leftarrow \theta_{t-1} - \gamma g_t \\
&\rule{110mm}{0.4pt} \\
&\textbf{return } \theta_t \\
&\rule{110mm}{0.4pt}
\end{aligned}
$$

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **maximize** (*bool, optional*) – maximize the params based on the objective, instead of minimizing (default: False)
- **foreach** (*bool, optional*) – whether foreach implementation of optimizer is used. If unspecified by the user (so foreach is None), we will try to use foreach over the for-loop implementation on CUDA, since it is usually significantly more performant. Note that the foreach implementation uses ~ sizeof(params) more peak memory than the for-loop version due to the intermediates being a tensorlist vs just one tensor. If memory is prohibitive, batch fewer parameters through the optimizer at a time or switch this flag to False (default: None)
- **differentiable** (*bool, optional*) – whether autograd should occur through the optimizer step in training. Otherwise, the step() function runs in a torch.no_grad() context. Setting to True can impair performance, so leave it False if you don't intend to run autograd through this instance (default: False)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

• NOTE

The implementation of SGD with Momentum/Nesterov subtly differs from Sutskever et. al. and implementations in some other frameworks.

Considering the specific case of Momentum, the update can be written as

$$
\begin{aligned}
v_{t+1} &= \mu * v_t + g_{t+1}, \\
p_{t+1} &= p_t - \text{lr} * v_{t+1},
\end{aligned}
$$

where p, g, v and μ denote the parameters, gradient, velocity, and momentum respectively.

This is in contrast to Sutskever et. al. and other frameworks which employ an update of the form

$$
\begin{aligned}
v_{t+1} &= \mu * v_t + \text{lr} * g_{t+1}, \\
p_{t+1} &= p_t - v_{t+1}.
\end{aligned}
$$

The Nesterov version is analogously modified.

Moreover, the initial value of the momentum buffer is set to the gradient value at the first step. This is in contrast to some other frameworks that initialize it to all zeros.

add_param_group(*param_group*)

Add a param group to the `Optimizer` s *param_groups*.

This can be useful when fine tuning a pre-trained network as frozen layers can be made trainable and added to the `Optimizer` as training progresses.

Parameters

param_group (*dict*) – Specifies what Tensors should be optimized along with group specific optimization options.

load_state_dict(*state_dict*)

Loads the optimizer state.

Parameters

state_dict (*dict*) – optimizer state. Should be an object returned from a call to `state_dict()`.

register_load_state_dict_post_hook(*hook*, *prepend=False*)

Register a load_state_dict post-hook which will be called after `load_state_dict()` is called. It should have the following signature:

```
hook(optimizer) -> None
```

The `optimizer` argument is the optimizer instance being used.

The hook will be called with argument `self` after calling `load_state_dict` on `self`. The registered hook can be used to perform post-processing after `load_state_dict` has loaded the `state_dict`.

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If True, the provided post `hook` will be fired before all the already registered post-hooks on `load_state_dict`. Otherwise, the provided `hook` will be fired after all the already registered post-hooks. (default: False)

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemoveableHandle`

register_load_state_dict_pre_hook(*hook*, *prepend=False*)

Register a load_state_dict pre-hook which will be called before `load_state_dict()` is called. It should have the following signature:

```
hook(optimizer, state_dict) -> state_dict or None
```

The `optimizer` argument is the optimizer instance being used and the `state_dict` argument is a shallow copy of the `state_dict` the user passed in to `load_state_dict`. The hook may modify the state_dict inplace or optionally return a new one. If a state_dict is returned, it will be used to be loaded into the optimizer.

The hook will be called with argument `self` and `state_dict` before calling `load_state_dict` on `self`. The registered hook can be used to perform pre-processing before the `load_state_dict` call is made.

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If True, the provided pre `hook` will be fired before all the already registered pre-hooks on `load_state_dict`. Otherwise, the provided `hook` will be fired after all the already registered pre-hooks. (default: False)

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemoveableHandle`

register_state_dict_post_hook(*hook*, *prepend=False*)

Register a state dict post-hook which will be called after `state_dict()` is called. It should have the following signature:

```
hook(optimizer, state_dict) -> state_dict or None
```

The hook will be called with arguments `self` and `state_dict` after generating a `state_dict` on `self`. The hook may modify the state_dict inplace or optionally return a new one. The registered hook can be used to perform post-processing on the `state_dict` before it is returned.

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If True, the provided post `hook` will be fired before all the already registered post-hooks on `state_dict`. Otherwise, the provided `hook`

will be fired after all the already registered post-hooks. (default: False)

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemoveableHandle`

register_state_dict_pre_hook(*hook*, *prepend=False*)

Register a state dict pre-hook which will be called before `state_dict()` is called. It should have the following signature:

```
hook(optimizer) -> None
```

The `optimizer` argument is the optimizer instance being used. The hook will be called with argument `self` before calling `state_dict` on `self`. The registered hook can be used to perform pre-processing before the `state_dict` call is made.

Parameters

- **hook** (*Callable*) – The user defined hook to be registered.
- **prepend** (*bool*) – If True, the provided pre `hook` will be fired before all the already registered pre-hooks on `state_dict`. Otherwise, the provided `hook` will be fired after all the already registered pre-hooks. (default: False)

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemoveableHandle`

register_step_post_hook(*hook*)

Register an optimizer step post hook which will be called after optimizer step. It should have the following signature:

```
hook(optimizer, args, kwargs) -> None
```

The `optimizer` argument is the optimizer instance being used.

Parameters

hook (*Callable*) – The user defined hook to be registered.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

register_step_pre_hook(*hook*)

Register an optimizer step pre hook which will be called before optimizer step. It should have the following signature:

```
hook(optimizer, args, kwargs) -> None or modified args and kwargs
```

The `optimizer` argument is the optimizer instance being used. If args and kwargs are modified by the pre-hook, then the transformed values are returned as a tuple containing the new_args and new_kwargs.

Parameters

hook (*Callable*) – The user defined hook to be registered.

Returns

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type

`torch.utils.hooks.RemovableHandle`

state_dict()

Returns the state of the optimizer as a `dict`.

It contains two entries:

- `state`: a Dict holding current optimization state. Its content

  differs between optimizer classes, but some common characteristics hold. For example, state is saved per parameter, and the parameter itself is NOT saved. `state` is a Dictionary mapping parameter ids to a Dict with state corresponding to each parameter.

- `param_groups`: a List containing all parameter groups where each

  parameter group is a Dict. Each parameter group contains metadata specific to the optimizer, such as learning rate and weight decay, as well as a List of parameter IDs of the parameters in the group.

NOTE: The parameter IDs may look like indices but they are just IDs associating state with param_group. When loading from a state_dict, the optimizer will zip the param_group `params` (int IDs) and the optimizer `param_groups` (actual `nn.Parameter` s) in order to match state WITHOUT additional verification.

A returned state dict might look something like:

```
{
    'state': {
        0: {'momentum_buffer': tensor(...), ...},
        1: {'momentum_buffer': tensor(...), ...},
        2: {'momentum_buffer': tensor(...), ...},
        3: {'momentum_buffer': tensor(...), ...}
    },
    'param_groups': [
        {
            'lr': 0.01,
            'weight_decay': 0,
            ...
            'params': [0]
        },
        {
            'lr': 0.001,
            'weight_decay': 0.5,
            ...
            'params': [1, 2, 3]
        }
    ]
}
```

Return type

Dict[str, *Any*]

step(*closure=None*) [SOURCE]

Performs a single optimization step.

Parameters

closure (*Callable, optional*) – A closure that reevaluates the model and returns the loss.

zero_grad(*set_to_none=True*)

Resets the gradients of all optimized `torch.Tensor` s.

Parameters

set_to_none (*bool*) – instead of setting to zero, set the grads to None. This will in general have lower memory footprint, and can modestly improve performance. However, it changes certain behaviors. For example: 1. When the user tries to access a gradient and perform manual ops on it, a None attribute or

TORCH.RANDPERM

torch.randperm(*n*, ***, *generator=None*, *out=None*, *dtype=torch.int64*, *layout=torch.strided*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor

Returns a random permutation of integers from `0` to `n - 1`.

Parameters

n (*int*) – the upper bound (exclusive)

Keyword Arguments

- **generator** (`torch.Generator` , optional) – a pseudorandom number generator for sampling
- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype` , optional) – the desired data type of returned tensor. Default: `torch.int64` .
- **layout** (`torch.layout` , optional) – the desired layout of returned Tensor. Default: `torch.strided` .
- **device** (`torch.device` , optional) – the desired device of returned tensor. Default: if `None` , uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False` .
- **pin_memory** (*bool, optional*) – If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False` .

Example:

```
>>> torch.randperm(4)
tensor([2, 1, 0, 3])
```

Previous

Next

TORCH.SIGN

torch.sign(*input*, ***, *out=None*) → Tensor

Returns a new tensor with the signs of the elements of `input`.

$$\text{out}_i = \text{sgn}(\text{input}_i)$$

Parameters

input (*Tensor*) – the input tensor.

Keyword Arguments

out (*Tensor*, *optional*) – the output tensor.

Example:

```
>>> a = torch.tensor([0.7, -1.2, 0., 2.3])
>>> a
tensor([ 0.7000, -1.2000,  0.0000,  2.3000])
>>> torch.sign(a)
tensor([ 1., -1.,  0.,  1.])
```

Previous

Next

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >

TORCH.SIN

torch.sin(*input*, *, *out=None*) → Tensor

Returns a new tensor with the sine of the elements of `input`.

$$\text{out}_i = \sin(\text{input}_i)$$

Parameters

input (*Tensor*) – the input tensor.

Keyword Arguments

out (*Tensor*, *optional*) – the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.5461,  0.1347, -2.7266, -0.2746])
>>> torch.sin(a)
tensor([-0.5194,  0.1343, -0.4032, -0.2711])
```

Previous

Next

TORCH.SQUEEZE

torch.squeeze(*input*, *dim=None*) → Tensor

Returns a tensor with all specified dimensions of `input` of size `1` removed.

For example, if *input* is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then the *input.squeeze*() will be of shape: $(A \times B \times C \times D)$.

When `dim` is given, a squeeze operation is done only in the given dimension(s). If *input* is of shape: $(A \times 1 \times B)$, `squeeze(input, 0)` leaves the tensor unchanged, but `squeeze(input, 1)` will squeeze the tensor to the shape $(A \times B)$.

• NOTE

The returned tensor shares the storage with the input tensor, so changing the contents of one will change the contents of the other.

• WARNING

If the tensor has a batch dimension of size 1, then *squeeze*(*input*) will also remove the batch dimension, which can lead to unexpected errors. Consider specifying only the dims you wish to be squeezed.

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int* or *tuple* of *ints*, *optional*) –
  if given, the input will be squeezed

  only in the specified dimensions.

  Changed in version 2.0: `dim` now accepts tuples of dimensions.

Example:

```
>>> x = torch.zeros(2, 1, 2, 1, 2)
>>> x.size()
torch.Size([2, 1, 2, 1, 2])
>>> y = torch.squeeze(x)
>>> y.size()
torch.Size([2, 2, 2])
>>> y = torch.squeeze(x, 0)
>>> y.size()
torch.Size([2, 1, 2, 1, 2])
>>> y = torch.squeeze(x, 1)
>>> y.size()
torch.Size([2, 2, 1, 2])
>>> y = torch.squeeze(x, (1, 2, 3))
torch.Size([2, 2, 2])
```

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >

TORCH.STACK

torch.stack(*tensors*, *dim=0*, ***, *out=None*) → Tensor

Concatenates a sequence of tensors along a new dimension.

All tensors need to be of the same size.

• SEE ALSO

`torch.cat()` concatenates the given sequence along an existing dimension.

Parameters

- **tensors** (*sequence of Tensors*) – sequence of tensors to concatenate
- **dim** (*int*) – dimension to insert. Has to be between 0 and the number of dimensions of concatenated tensors (inclusive)

Keyword Arguments

out (*Tensor, optional*) – the output tensor.