# `copy` — Shallow and deep copy operations

**Source code:** Lib/copy.py

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

`copy.copy(obj)`

Return a shallow copy of *obj*.

`copy.deepcopy(obj[, memo])`

Return a deep copy of *obj*.

`copy.replace(obj, /, **changes)`

Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

*Added in version 3.13.*

`exception copy.Error`

Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The `deepcopy()` function avoids these problems by:

- keeping a `memo` dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the `pickle` module.

Shallow copies of dictionaries can be made using `dict.copy()`, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module `pickle` for information on these methods. In fact, the `copy` module uses the registered pickle functions from the `copyreg` module.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`.

`object.`**`__copy__`**`(`*`self`*`)`

Called to implement the shallow copy operation; no additional arguments are passed.

`object.`**`__deepcopy__`**`(`*`self, memo`*`)`

Called to implement the deep copy operation; it is passed one argument, the *memo* dictionary. If the __deepcopy__ implementation needs to make a deep copy of a component, it should call the `deepcopy()` function with the component as first argument and the *memo* dictionary as second argument. The *memo* dictionary should be treated as an opaque object.

Function `copy.replace()` is more limited than `copy()` and `deepcopy()`, and only supports named tuples created by `namedtuple()`, `dataclasses`, and other classes which define method `__replace__()`.

`object.`**`__replace__`**`(`*`self`*`, /, `*`**changes`*`)`

This method should create a new object of the same type, replacing fields with values from *changes*.

**See also:**

**Module** `pickle`

Discussion of the special methods used to support object state retrieval and restoration.