```python
class SimpleNamespace:
    def __init__(self, /, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,
→SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

SimpleNamespace may be useful as a replacement for class NS: pass. However, for a structured record type use *namedtuple()* instead.

New in version 3.3.

Changed in version 3.9: Attribute order in the repr changed from alphabetical to insertion (like dict).

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's __getattr__ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see *enum.Enum* for an example).

New in version 3.4.

8.10.4 Coroutine Utility Functions

types.**coroutine**(*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the __await__() method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of *collections.abc. Generator*, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

New in version 3.5.

8.11 `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**(*x*)

> Return a shallow copy of *x*.

copy.**deepcopy**(*x*[, *memo*])

> Return a deep copy of *x*.

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__(). The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is 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.

See also:

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

8.12 `pprint` — Data pretty printer

Source code: Lib/pprint.py

The *pprint* module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.