8.9. `types` — Dynamic type creation and names for built-in types

Source code: Lib/types.py

This module defines utility function to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like `int` or `str` are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.9.1. Dynamic Type Creation

`types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: ns`.

New in version 3.3.

`types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

New in version 3.3.

Changed in version 3.6: The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method,

See also:

Metaclasses
: Full details of the class creation process supported by these functions

PEP 3115 - Metaclasses in Python 3000
: Introduced the `__prepare__` namespace hook

8.9.2. Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for `isinstance()` or `issubclass()` checks.

Standard names are defined for the following types:

`types.`**`FunctionType`**
`types.`**`LambdaType`**

The type of user-defined functions and functions created by `lambda` expressions.

`types.`**`GeneratorType`**

The type of generator-iterator objects, created by generator functions.

`types.`**`CoroutineType`**

The type of coroutine objects, created by `async def` functions.

New in version 3.5.

`types.`**`AsyncGeneratorType`**

The type of asynchronous generator-iterator objects, created by asynchronous generator functions.

New in version 3.6.

`types.`**`CodeType`**

The type for code objects such as returned by `compile()`.

`types.`**`MethodType`**

The type of methods of user-defined class instances.

`types.`**`BuiltinFunctionType`**
`types.`**`BuiltinMethodType`**

The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term “built-in” means “written in C”.)

class `types.`**`ModuleType`**(*name*, *doc=None*)

The type of modules. Constructor takes the name of the module to be created and optionally its docstring.

> **Note:** Use `importlib.util.module_from_spec()` to create a new module if you wish to set the various import-controlled attributes.

`__doc__`

The docstring of the module. Defaults to `None`.

`__loader__`

The loader which loaded the module. Defaults to `None`.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

`__name__`

The name of the module.

`__package__`

Which package a module belongs to. If the module is top-level (i.e. not a part of any specific package) then the attribute should be set to `''`, else it should be set to the name of the package (which can be `__name__` if the module is a package itself). Defaults to `None`.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

`types.`**`TracebackType`**

The type of traceback objects such as found in `sys.exc_info()[2]`.

`types.`**`FrameType`**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

`types.`**`GetSetDescriptorType`**

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the `property` type, but for classes defined in extension modules.

types.**MemberDescriptorType**

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the `property` type, but for classes defined in extension modules.

CPython implementation detail: In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class types.**MappingProxyType**(*mapping*)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

New in version 3.3.

key in proxy

Return `True` if the underlying mapping has a key *key*, else `False`.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a `KeyError` if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(*key*[, *default*])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return a new view of the underlying mapping's items (`(key, value)` pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

8.9.3. Additional Utility Classes and Functions

class `types.`**`SimpleNamespace`**

A simple `object` subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike `object`, with `SimpleNamespace` you can add and remove attributes. If a `SimpleNamespace` object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code:

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

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in ke
        return "{}({})".format(type(self).__name__, ", ".join(item

    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

`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.

`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 for an example).

New in version 3.4.

8.9.4. 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.