```
Spezielle Bezeichner (Identifier)          (C) 2017-2020 T.Birnthaler OSTC GmbH
================================

Doku --> http://docs.python.org/dev/peps/pep-0008
         http://www.pep8.org

Folgende Namenskonventionen fÃ¼r Bezeichner (Identifier) gelten in Python:

+----------+-------------------------------------------------------------------+
| Name     | Bedeutung                                                         |
+----------+-------------------------------------------------------------------+
| __*__    | INTERNE Namen (reserviert fÃ¼r Python selbst)                      |
| __*      | Private Namen von Klassen (mangled --> _CLASS__*)                 |
| _*       | Protected Namen von Klassen (fÃ¼r Vererbung)                       |
|          | (von "from MODULE import *" nicht importiert)                     |
+----------+-------------------------------------------------------------------+
| *_       | SchlÃ¼sselworte als Bezeichner (z.B. "if_")                        |
| _        | Im interaktiven Interpreter: Ergebnis der letzten Auswertung      |
|          | TemporÃ¤re Variable in Schleifen (for _ in ...)                    |
|          | "Wegwerfvariable" (z.B. *_ in Parameterleiste von Funktion)       |
|          | Internationalisierung (i18n, gettext)                             |
+----------+-------------------------------------------------------------------+
| self     | Objekt in normalen Methoden von Klassen                           |
| other    | 2. Objekt in normalen 2-wertigen Methoden von Klassen             |
| cls      | Klasse in Klassen-Methoden von Klassen                            |
| *args    | Positionsparameter-Sammler in "variadischen" Funktionen           |
| **kwargs | Keywordparameter-Sammler in "variadischen" Funktionen             |
+----------+-------------------------------------------------------------------+
| NAME     | Konstante       (nur GROSSb., Unterstrich, Ziffern)               |
| Name     | Klassenname     (Camelcase)                          Substantiv   |
| Name     | Exceptionklasse (Camelcase)                          Substantiv   |
| name     | Modulname       (kleinb., KEIN Unterstrich, Ziffern)              |
| name     | Variablename    (kleinb., Unterstr., Zif.)    Adjektiv/Substantiv |
| name     | Funkionsname    (kleinb., Unterstr., Zif.)                  Verb  |
| name     | Objektname      (kleinb., Unterstr., Zif.)    Adjektiv/Substantiv |
+----------+-------------------------------------------------------------------+

Qualifizierte Namen werden durch "." getrennt:

  MODUL.VARIABLE
  MODUL.FUNKTION()
  MODUL.SUBMODUL.VARIABLE
  MODUL.SUBMODUL.FUNKTION()


Spezielle Attribute
-------------------
Spezielle read-only Attribute, werden teilweise von der built-in Funktion
"dict()" nicht aufgelistet.

+------------------+-------------------------------------------------------+
| Attribut         | Beschreibung                                          |
+------------------+-------------------------------------------------------+
| __doc__          | Dokumentationsstring ("Docstring")                    |
| __name__         | Name von Klasse/Funktion/Methode/Descriptor/Generator |
| __qualname__     | Qualifizierter Name von ...                           |
| __defaults__     | Tuple mit Defaultwerten fÃ¼r Position-Parameter        |
| __code__         | Codeobjekt einer Funktion                             |
| __globals__      | Referenz zu Dictionary mit seinen globalen Variablen  |
| __dict__         | Speicher fÃ¼r dynamische Objektattribute (Name + Wert) |
| __slots__        | Speicher fÃ¼r statische Attributnamen von Objekten     |
| __closure__      | Bindungen freier Variablen an Werte                   |
| __annotations__  | Dictionary mit "Annotations"                          |
| __kwdefaults__   | Dictionary mit Defaultwerten fÃ¼r Keyword-Parameter    |
| __class__        | Klasse zu der eine Instanz gehÃ¶rt                     |
| __bases__        | Tupel der Basisklassen eines Klassenobjekts           |
| __mro__          | Tupel der Basisklassen fÃ¼r MethodenauflÃ¶sung          |
| mro()            | Reihenfolge der MethodenauflÃ¶sung                     |
| __subclasses__() | Liste schwacher Referenzen zu direkten Unterklassen   |
+------------------+-------------------------------------------------------+


Spezielle Methoden
------------------

Basic customization
===================

object.__new__(cls[, ...])

   Called to create a new instance of class *cls*.  "__new__()" is a
   static method (special-cased so you need not declare it as such)
```

```
   that takes the class of which an instance was requested as its
   first argument.  The remaining arguments are those passed to the
   object constructor expression (the call to the class).  The return
   value of "__new__()" should be the new object instance (usually an
   instance of *cls*).

   Typical implementations create a new instance of the class by
   invoking the superclassâM-^@M-^Ys "__new__()" method using
   "super().__new__(cls[, ...])" with appropriate arguments and then
   modifying the newly-created instance as necessary before returning
   it.

   If "__new__()" returns an instance of *cls*, then the new
   instanceâM-^@M-^Ys "__init__()" method will be invoked like
   "__init__(self[, ...])", where *self* is the new instance and the
   remaining arguments are the same as were passed to "__new__()".

   If "__new__()" does not return an instance of *cls*, then the new
   instanceâM-^@M-^Ys "__init__()" method will not be invoked.

   "__new__()" is intended mainly to allow subclasses of immutable
   types (like int, str, or tuple) to customize instance creation.  It
   is also commonly overridden in custom metaclasses in order to
   customize class creation.

object.__init__(self[, ...])

   Called after the instance has been created (by "__new__()"), but
   before it is returned to the caller.  The arguments are those
   passed to the class constructor expression.  If a base class has an
   "__init__()" method, the derived classâM-^@M-^Ys "__init__()" method, if
   any, must explicitly call it to ensure proper initialization of the
   base class part of the instance; for example:
   "super().__init__([args...])".

   Because "__new__()" and "__init__()" work together in constructing
   objects ("__new__()" to create it, and "__init__()" to customize
   it), no non-"None" value may be returned by "__init__()"; doing so
   will cause a "TypeError" to be raised at runtime.

object.__del__(self)

   Called when the instance is about to be destroyed.  This is also
   called a finalizer or (improperly) a destructor.  If a base class
   has a "__del__()" method, the derived classâM-^@M-^Ys "__del__()" method,
   if any, must explicitly call it to ensure proper deletion of the
   base class part of the instance.

   It is possible (though not recommended!) for the "__del__()" method
   to postpone destruction of the instance by creating a new reference
   to it.  This is called object *resurrection*.  It is
   implementation-dependent whether "__del__()" is called a second
   time when a resurrected object is about to be destroyed; the
   current *CPython* implementation only calls it once.

   It is not guaranteed that "__del__()" methods are called for
   objects that still exist when the interpreter exits.

   Note: "del x" doesnâM-^@M-^Yt directly call "x.__del__()" âM-^@M-^T the former
     decrements the reference count for "x" by one, and the latter is
     only called when "x"âM-^@M-^Ys reference count reaches zero.

   **CPython implementation detail:** It is possible for a reference
   cycle to prevent the reference count of an object from going to
   zero.  In this case, the cycle will be later detected and deleted
   by the *cyclic garbage collector*.  A common cause of reference
   cycles is when an exception has been caught in a local variable.
   The frameâM-^@M-^Ys locals then reference the exception, which references
   its own traceback, which references the locals of all frames caught
   in the traceback.

   See also: Documentation for the "gc" module.

   Warning: Due to the precarious circumstances under which
     "__del__()" methods are invoked, exceptions that occur during
     their execution are ignored, and a warning is printed to
     "sys.stderr" instead. In particular:

     * "__del__()" can be invoked when arbitrary code is being
       executed, including from any arbitrary thread.  If "__del__()"
       needs to take a lock or invoke any other blocking resource, it
       may deadlock as the resource may already be taken by the code
```

```
       that gets interrupted to execute "__del__()".

     * "__del__()" can be executed during interpreter shutdown.  As
       a consequence, the global variables it needs to access
       (including other modules) may already have been deleted or set
       to "None". Python guarantees that globals whose name begins
       with a single underscore are deleted from their module before
       other globals are deleted; if no other references to such
       globals exist, this may help in assuring that imported modules
       are still available at the time when the "__del__()" method is
       called.

object.__repr__(self)

   Called by the "repr()" built-in function to compute the âM-^@M-^\officialâM-^@M-^]
   string representation of an object.  If at all possible, this
   should look like a valid Python expression that could be used to
   recreate an object with the same value (given an appropriate
   environment).  If this is not possible, a string of the form
   "<...some useful description...>" should be returned. The return
   value must be a string object. If a class defines "__repr__()" but
   not "__str__()", then "__repr__()" is also used when an âM-^@M-^\informalâM-^@M-^]
   string representation of instances of that class is required.

   This is typically used for debugging, so it is important that the
   representation is information-rich and unambiguous.

object.__str__(self)

   Called by "str(object)" and the built-in functions "format()" and
   "print()" to compute the âM-^@M-^\informalâM-^@M-^] or nicely printable string
   representation of an object.  The return value must be a string
   object.

   This method differs from "object.__repr__()" in that there is no
   expectation that "__str__()" return a valid Python expression: a
   more convenient or concise representation can be used.

   The default implementation defined by the built-in type "object"
   calls "object.__repr__()".

object.__bytes__(self)

   Called by bytes to compute a byte-string representation of an
   object. This should return a "bytes" object.

object.__format__(self, format_spec)

   Called by the "format()" built-in function, and by extension,
   evaluation of formatted string literals and the "str.format()"
   method, to produce a âM-^@M-^\formattedâM-^@M-^] string representation of an
   object. The "format_spec" argument is a string that contains a
   description of the formatting options desired. The interpretation
   of the "format_spec" argument is up to the type implementing
   "__format__()", however most classes will either delegate
   formatting to one of the built-in types, or use a similar
   formatting option syntax.

   See Format Specification Mini-Language for a description of the
   standard formatting syntax.

   The return value must be a string object.

   Changed in version 3.4: The __format__ method of "object" itself
   raises a "TypeError" if passed any non-empty string.

   Changed in version 3.7: "object.__format__(x, '')" is now
   equivalent to "str(x)" rather than "format(str(self), '')".

object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)

   These are the so-called âM-^@M-^\rich comparisonâM-^@M-^] methods. The
   correspondence between operator symbols and method names is as
   follows: "x<y" calls "x.__lt__(y)", "x<=y" calls "x.__le__(y)",
   "x==y" calls "x.__eq__(y)", "x!=y" calls "x.__ne__(y)", "x>y" calls
   "x.__gt__(y)", and "x>=y" calls "x.__ge__(y)".
```

```
   A rich comparison method may return the singleton "NotImplemented"
   if it does not implement the operation for a given pair of
   arguments. By convention, "False" and "True" are returned for a
   successful comparison. However, these methods can return any value,
   so if the comparison operator is used in a Boolean context (e.g.,
   in the condition of an "if" statement), Python will call "bool()"
   on the value to determine if the result is true or false.

   By default, "__ne__()" delegates to "__eq__()" and inverts the
   result unless it is "NotImplemented".  There are no other implied
   relationships among the comparison operators, for example, the
   truth of "(x<y or x==y)" does not imply "x<=y". To automatically
   generate ordering operations from a single root operation, see
   "functools.total_ordering()".

   See the paragraph on "__hash__()" for some important notes on
   creating *hashable* objects which support custom comparison
   operations and are usable as dictionary keys.

   There are no swapped-argument versions of these methods (to be used
   when the left argument does not support the operation but the right
   argument does); rather, "__lt__()" and "__gt__()" are each otherâM-^@M-^Ys
   reflection, "__le__()" and "__ge__()" are each otherâM-^@M-^Ys reflection,
   and "__eq__()" and "__ne__()" are their own reflection. If the
   operands are of different types, and right operandâM-^@M-^Ys type is a
   direct or indirect subclass of the left operandâM-^@M-^Ys type, the
   reflected method of the right operand has priority, otherwise the
   left operandâM-^@M-^Ys method has priority.  Virtual subclassing is not
   considered.

object.__hash__(self)

   Called by built-in function "hash()" and for operations on members
   of hashed collections including "set", "frozenset", and "dict".
   "__hash__()" should return an integer. The only required property
   is that objects which compare equal have the same hash value; it is
   advised to mix together the hash values of the components of the
   object that also play a part in comparison of objects by packing
   them into a tuple and hashing the tuple. Example:

      def __hash__(self):
          return hash((self.name, self.nick, self.color))

   Note: "hash()" truncates the value returned from an objectâM-^@M-^Ys
     custom "__hash__()" method to the size of a "Py_ssize_t".  This
     is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit
     builds. If an objectâM-^@M-^Ys   "__hash__()" must interoperate on builds
     of different bit sizes, be sure to check the width on all
     supported builds.  An easy way to do this is with "python -c
     "import sys; print(sys.hash_info.width)"".

   If a class does not define an "__eq__()" method it should not
   define a "__hash__()" operation either; if it defines "__eq__()"
   but not "__hash__()", its instances will not be usable as items in
   hashable collections.  If a class defines mutable objects and
   implements an "__eq__()" method, it should not implement
   "__hash__()", since the implementation of hashable collections
   requires that a keyâM-^@M-^Ys hash value is immutable (if the objectâM-^@M-^Ys hash
   value changes, it will be in the wrong hash bucket).

   User-defined classes have "__eq__()" and "__hash__()" methods by
   default; with them, all objects compare unequal (except with
   themselves) and "x.__hash__()" returns an appropriate value such
   that "x == y" implies both that "x is y" and "hash(x) == hash(y)".

   A class that overrides "__eq__()" and does not define "__hash__()"
   will have its "__hash__()" implicitly set to "None".  When the
   "__hash__()" method of a class is "None", instances of the class
   will raise an appropriate "TypeError" when a program attempts to
   retrieve their hash value, and will also be correctly identified as
   unhashable when checking "isinstance(obj,
   collections.abc.Hashable)".

   If a class that overrides "__eq__()" needs to retain the
   implementation of "__hash__()" from a parent class, the interpreter
   must be told this explicitly by setting "__hash__ =
   <ParentClass>.__hash__".

   If a class that does not override "__eq__()" wishes to suppress
   hash support, it should include "__hash__ = None" in the class
   definition. A class which defines its own "__hash__()" that
   explicitly raises a "TypeError" would be incorrectly identified as
```

```
   hashable by an "isinstance(obj, collections.abc.Hashable)" call.

   Note: By default, the "__hash__()" values of str, bytes and
     datetime objects are âM-^@M-^\saltedâM-^@M-^] with an unpredictable random value.
     Although they remain constant within an individual Python
     process, they are not predictable between repeated invocations of
     Python.This is intended to provide protection against a denial-
     of-service caused by carefully-chosen inputs that exploit the
     worst case performance of a dict insertion, O(n^2) complexity.
     See http://www.ocert.org/advisories/ocert-2011-003.html for
     details.Changing hash values affects the iteration order of
     dicts, sets and other mappings.  Python has never made guarantees
     about this ordering (and it typically varies between 32-bit and
     64-bit builds).See also "PYTHONHASHSEED".

   Changed in version 3.3: Hash randomization is enabled by default.

object.__bool__(self)

   Called to implement truth value testing and the built-in operation
   "bool()"; should return "False" or "True".  When this method is not
   defined, "__len__()" is called, if it is defined, and the object is
   considered true if its result is nonzero.  If a class defines
   neither "__len__()" nor "__bool__()", all its instances are
   considered true.


Customizing attribute access
============================

The following methods can be defined to customize the meaning of
attribute access (use of, assignment to, or deletion of "x.name") for
class instances.

object.__getattr__(self, name)

   Called when the default attribute access fails with an
   "AttributeError" (either "__getattribute__()" raises an
   "AttributeError" because *name* is not an instance attribute or an
   attribute in the class tree for "self"; or "__get__()" of a *name*
   property raises "AttributeError").  This method should either
   return the (computed) attribute value or raise an "AttributeError"
   exception.

   Note that if the attribute is found through the normal mechanism,
   "__getattr__()" is not called.  (This is an intentional asymmetry
   between "__getattr__()" and "__setattr__()".) This is done both for
   efficiency reasons and because otherwise "__getattr__()" would have
   no way to access other attributes of the instance.  Note that at
   least for instance variables, you can fake total control by not
   inserting any values in the instance attribute dictionary (but
   instead inserting them in another object).  See the
   "__getattribute__()" method below for a way to actually get total
   control over attribute access.

object.__getattribute__(self, name)

   Called unconditionally to implement attribute accesses for
   instances of the class. If the class also defines "__getattr__()",
   the latter will not be called unless "__getattribute__()" either
   calls it explicitly or raises an "AttributeError". This method
   should return the (computed) attribute value or raise an
   "AttributeError" exception. In order to avoid infinite recursion in
   this method, its implementation should always call the base class
   method with the same name to access any attributes it needs, for
   example, "object.__getattribute__(self, name)".

   Note: This method may still be bypassed when looking up special
     methods as the result of implicit invocation via language syntax
     or built-in functions. See Special method lookup.

object.__setattr__(self, name, value)

   Called when an attribute assignment is attempted.  This is called
   instead of the normal mechanism (i.e. store the value in the
   instance dictionary). *name* is the attribute name, *value* is the
   value to be assigned to it.

   If "__setattr__()" wants to assign to an instance attribute, it
   should call the base class method with the same name, for example,
   "object.__setattr__(self, name, value)".
```

```
object.__delattr__(self, name)

   Like "__setattr__()" but for attribute deletion instead of
   assignment.  This should only be implemented if "del obj.name" is
   meaningful for the object.

object.__dir__(self)

   Called when "dir()" is called on the object. A sequence must be
   returned. "dir()" converts the returned sequence to a list and
   sorts it.


Customizing module attribute access
-----------------------------------

Special names "__getattr__" and "__dir__" can be also used to
customize access to module attributes. The "__getattr__" function at
the module level should accept one argument which is the name of an
attribute and return the computed value or raise an "AttributeError".
If an attribute is not found on a module object through the normal
lookup, i.e. "object.__getattribute__()", then "__getattr__" is
searched in the module "__dict__" before raising an "AttributeError".
If found, it is called with the attribute name and the result is
returned.

The "__dir__" function should accept no arguments, and return a list
of strings that represents the names accessible on module. If present,
this function overrides the standard "dir()" search on a module.

For a more fine grained customization of the module behavior (setting
attributes, properties, etc.), one can set the "__class__" attribute
of a module object to a subclass of "types.ModuleType". For example:

   import sys
   from types import ModuleType

   class VerboseModule(ModuleType):
       def __repr__(self):
           return f'Verbose {self.__name__}'

       def __setattr__(self, attr, value):
           print(f'Setting {attr}...')
           setattr(self, attr, value)

   sys.modules[__name__].__class__ = VerboseModule

Note: Defining module "__getattr__" and setting module "__class__"
  only affect lookups made using the attribute access syntax âM-^@M-^S
  directly accessing the module globals (whether by code within the
  module, or via a reference to the moduleâM-^@M-^Ys globals dictionary) is
  unaffected.

Changed in version 3.5: "__class__" module attribute is now writable.

New in version 3.7: "__getattr__" and "__dir__" module attributes.

See also:

  **PEP 562** - Module __getattr__ and __dir__
     Describes the "__getattr__" and "__dir__" functions on modules.


Implementing Descriptors
------------------------

The following methods only apply when an instance of the class
containing the method (a so-called *descriptor* class) appears in an
*owner* class (the descriptor must be in either the ownerâM-^@M-^Ys class
dictionary or in the class dictionary for one of its parents).  In the
examples below, âM-^@M-^\the attributeâM-^@M-^] refers to the attribute whose name is
the key of the property in the owner classâM-^@M-^Y "__dict__".

object.__get__(self, instance, owner)

   Called to get the attribute of the owner class (class attribute
   access) or of an instance of that class (instance attribute
   access). *owner* is always the owner class, while *instance* is the
   instance that the attribute was accessed through, or "None" when
   the attribute is accessed through the *owner*.  This method should
   return the (computed) attribute value or raise an "AttributeError"
   exception.
```

```
object.__set__(self, instance, value)

   Called to set the attribute on an instance *instance* of the owner
   class to a new value, *value*.

object.__delete__(self, instance)

   Called to delete the attribute on an instance *instance* of the
   owner class.

object.__set_name__(self, owner, name)

   Called at the time the owning class *owner* is created. The
   descriptor has been assigned to *name*.

   New in version 3.6.

The attribute "__objclass__" is interpreted by the "inspect" module as
specifying the class where this object was defined (setting this
appropriately can assist in runtime introspection of dynamic class
attributes). For callables, it may indicate that an instance of the
given type (or a subclass) is expected or required as the first
positional argument (for example, CPython sets this attribute for
unbound methods that are implemented in C).


Invoking Descriptors
--------------------

In general, a descriptor is an object attribute with âM-^@M-^\binding
behaviorâM-^@M-^], one whose attribute access has been overridden by methods
in the descriptor protocol:  "__get__()", "__set__()", and
"__delete__()". If any of those methods are defined for an object, it
is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete
the attribute from an objectâM-^@M-^Ys dictionary. For instance, "a.x" has a
lookup chain starting with "a.__dict__['x']", then
"type(a).__dict__['x']", and continuing through the base classes of
"type(a)" excluding metaclasses.

However, if the looked-up value is an object defining one of the
descriptor methods, then Python may override the default behavior and
invoke the descriptor method instead.  Where this occurs in the
precedence chain depends on which descriptor methods were defined and
how they were called.

The starting point for descriptor invocation is a binding, "a.x". How
the arguments are assembled depends on "a":

Direct Call
   The simplest and least common call is when user code directly
   invokes a descriptor method:    "x.__get__(a)".

Instance Binding
   If binding to an object instance, "a.x" is transformed into the
   call: "type(a).__dict__['x'].__get__(a, type(a))".

Class Binding
   If binding to a class, "A.x" is transformed into the call:
   "A.__dict__['x'].__get__(None, A)".

Super Binding
   If "a" is an instance of "super", then the binding "super(B,
   obj).m()" searches "obj.__class__.__mro__" for the base class "A"
   immediately preceding "B" and then invokes the descriptor with the
   call: "A.__dict__['m'].__get__(obj, obj.__class__)".

For instance bindings, the precedence of descriptor invocation depends
on the which descriptor methods are defined.  A descriptor can define
any combination of "__get__()", "__set__()" and "__delete__()".  If it
does not define "__get__()", then accessing the attribute will return
the descriptor object itself unless there is a value in the objectâM-^@M-^Ys
instance dictionary.  If the descriptor defines "__set__()" and/or
"__delete__()", it is a data descriptor; if it defines neither, it is
a non-data descriptor.  Normally, data descriptors define both
"__get__()" and "__set__()", while non-data descriptors have just the
"__get__()" method.  Data descriptors with "__set__()" and "__get__()"
defined always override a redefinition in an instance dictionary.  In
contrast, non-data descriptors can be overridden by instances.
```

```
Python methods (including "staticmethod()" and "classmethod()") are
implemented as non-data descriptors.  Accordingly, instances can
redefine and override methods.  This allows individual instances to
acquire behaviors that differ from other instances of the same class.

The "property()" function is implemented as a data descriptor.
Accordingly, instances cannot override the behavior of a property.


__slots__
---------

*__slots__* allow us to explicitly declare data members (like
properties) and deny the creation of *__dict__* and *__weakref__*
(unless explicitly declared in *__slots__* or available in a parent.)

The space saved over using *__dict__* can be significant.

object.__slots__

   This class variable can be assigned a string, iterable, or sequence
   of strings with variable names used by instances.  *__slots__*
   reserves space for the declared variables and prevents the
   automatic creation of *__dict__* and *__weakref__* for each
   instance.


Notes on using *__slots__*
~~~~~~~~~~~~~~~~~~~~~~~~~~

* When inheriting from a class without *__slots__*, the *__dict__*
  and *__weakref__* attribute of the instances will always be
  accessible.

* Without a *__dict__* variable, instances cannot be assigned new
  variables not listed in the *__slots__* definition.  Attempts to
  assign to an unlisted variable name raises "AttributeError". If
  dynamic assignment of new variables is desired, then add
  "'__dict__'" to the sequence of strings in the *__slots__*
  declaration.

* Without a *__weakref__* variable for each instance, classes
  defining *__slots__* do not support weak references to its
  instances. If weak reference support is needed, then add
  "'__weakref__'" to the sequence of strings in the *__slots__*
  declaration.

* *__slots__* are implemented at the class level by creating
  descriptors (Implementing Descriptors) for each variable name.  As a
  result, class attributes cannot be used to set default values for
  instance variables defined by *__slots__*; otherwise, the class
  attribute would overwrite the descriptor assignment.

* The action of a *__slots__* declaration is not limited to the
  class where it is defined.  *__slots__* declared in parents are
  available in child classes. However, child subclasses will get a
  *__dict__* and *__weakref__* unless they also define *__slots__*
  (which should only contain names of any *additional* slots).

* If a class defines a slot also defined in a base class, the
  instance variable defined by the base class slot is inaccessible
  (except by retrieving its descriptor directly from the base class).
  This renders the meaning of the program undefined.  In the future, a
  check may be added to prevent this.

* Nonempty *__slots__* does not work for classes derived from
  âM-^@M-^\variable-lengthâM-^@M-^] built-in types such as "int", "bytes" and "tuple".

* Any non-string iterable may be assigned to *__slots__*. Mappings
  may also be used; however, in the future, special meaning may be
  assigned to the values corresponding to each key.

* *__class__* assignment works only if both classes have the same
  *__slots__*.

* Multiple inheritance with multiple slotted parent classes can be
  used, but only one parent is allowed to have attributes created by
  slots (the other bases must have empty slot layouts) - violations
  raise "TypeError".


Customizing class creation
```

```
=========================

Whenever a class inherits from another class, *__init_subclass__* is
called on that class. This way, it is possible to write classes which
change the behavior of subclasses. This is closely related to class
decorators, but where class decorators only affect the specific class
theyâM-^@M-^Yre applied to, "__init_subclass__" solely applies to future
subclasses of the class defining the method.

classmethod object.__init_subclass__(cls)

   This method is called whenever the containing class is subclassed.
   *cls* is then the new subclass. If defined as a normal instance
   method, this method is implicitly converted to a class method.

   Keyword arguments which are given to a new class are passed to the
   parentâM-^@M-^Ys class "__init_subclass__". For compatibility with other
   classes using "__init_subclass__", one should take out the needed
   keyword arguments and pass the others over to the base class, as
   in:

      class Philosopher:
          def __init_subclass__(cls, default_name, **kwargs):
              super().__init_subclass__(**kwargs)
              cls.default_name = default_name

      class AustralianPhilosopher(Philosopher, default_name="Bruce"):
          pass

   The default implementation "object.__init_subclass__" does nothing,
   but raises an error if it is called with any arguments.

   Note: The metaclass hint "metaclass" is consumed by the rest of
     the type machinery, and is never passed to "__init_subclass__"
     implementations. The actual metaclass (rather than the explicit
     hint) can be accessed as "type(cls)".

   New in version 3.6.


Metaclasses
-----------

By default, classes are constructed using "type()". The class body is
executed in a new namespace and the class name is bound locally to the
result of "type(name, bases, namespace)".

The class creation process can be customized by passing the
"metaclass" keyword argument in the class definition line, or by
inheriting from an existing class that included such an argument. In
the following example, both "MyClass" and "MySubclass" are instances
of "Meta":

   class Meta(type):
       pass

   class MyClass(metaclass=Meta):
       pass

   class MySubclass(MyClass):
       pass

Any other keyword arguments that are specified in the class definition
are passed through to all metaclass operations described below.

When a class definition is executed, the following steps occur:

* MRO entries are resolved

* the appropriate metaclass is determined

* the class namespace is prepared

* the class body is executed

* the class object is created


Resolving MRO entries
---------------------

If a base that appears in class definition is not an instance of
```

```
"type", then an "__mro_entries__" method is searched on it. If found,
it is called with the original bases tuple. This method must return a
tuple of classes that will be used instead of this base. The tuple may
be empty, in such case the original base is ignored.

See also: **PEP 560** - Core support for typing module and generic
  types


Determining the appropriate metaclass
-------------------------------------

The appropriate metaclass for a class definition is determined as
follows:

* if no bases and no explicit metaclass are given, then "type()" is
  used

* if an explicit metaclass is given and it is *not* an instance of
  "type()", then it is used directly as the metaclass

* if an instance of "type()" is given as the explicit metaclass, or
  bases are defined, then the most derived metaclass is used

The most derived metaclass is selected from the explicitly specified
metaclass (if any) and the metaclasses (i.e. "type(cls)") of all
specified base classes. The most derived metaclass is one which is a
subtype of *all* of these candidate metaclasses. If none of the
candidate metaclasses meets that criterion, then the class definition
will fail with "TypeError".


Preparing the class namespace
-----------------------------

Once the appropriate metaclass has been identified, then the class
namespace is prepared. If the metaclass has a "__prepare__" attribute,
it is called as "namespace = metaclass.__prepare__(name, bases,
**kwds)" (where the additional keyword arguments, if any, come from
the class definition).

If the metaclass has no "__prepare__" attribute, then the class
namespace is initialised as an empty ordered mapping.

See also:

  **PEP 3115** - Metaclasses in Python 3000
     Introduced the "__prepare__" namespace hook


Executing the class body
------------------------

The class body is executed (approximately) as "exec(body, globals(),
namespace)". The key difference from a normal call to "exec()" is that
lexical scoping allows the class body (including any methods) to
reference names from the current and outer scopes when the class
definition occurs inside a function.

However, even when the class definition occurs inside the function,
methods defined inside the class still cannot see names defined at the
class scope. Class variables must be accessed through the first
parameter of instance or class methods, or through the implicit
lexically scoped "__class__" reference described in the next section.


Creating the class object
-------------------------

Once the class namespace has been populated by executing the class
body, the class object is created by calling "metaclass(name, bases,
namespace, **kwds)" (the additional keywords passed here are the same
as those passed to "__prepare__").

This class object is the one that will be referenced by the zero-
argument form of "super()". "__class__" is an implicit closure
reference created by the compiler if any methods in a class body refer
to either "__class__" or "super". This allows the zero argument form
of "super()" to correctly identify the class being defined based on
lexical scoping, while the class or instance that was used to make the
current call is identified based on the first argument passed to the
method.
```

CPython implementation detail: In CPython 3.6 and later, the
"__class__" cell is passed to the metaclass as a "__classcell__" entry
in the class namespace. If present, this must be propagated up to the
"type.__new__" call in order for the class to be initialised
correctly. Failing to do so will result in a "DeprecationWarning" in
Python 3.6, and a "RuntimeError" in Python 3.8.

When using the default metaclass "type", or any metaclass that
ultimately calls "type.__new__", the following additional
customisation steps are invoked after creating the class object:

* first, "type.__new__" collects all of the descriptors in the class
  namespace that define a "__set_name__()" method;

* second, all of these "__set_name__" methods are called with the
  class being defined and the assigned name of that particular
  descriptor; and

* finally, the "__init_subclass__()" hook is called on the immediate
  parent of the new class in its method resolution order.

After the class object is created, it is passed to the class
decorators included in the class definition (if any) and the resulting
object is bound in the local namespace as the defined class.

When a new class is created by "type.__new__", the object provided as
the namespace parameter is copied to a new ordered mapping and the
original object is discarded. The new copy is wrapped in a read-only
proxy, which becomes the "__dict__" attribute of the class object.

See also:

  PEP 3135 - New super
     Describes the implicit "__class__" closure reference

Metaclass example

The potential uses for metaclasses are boundless. Some ideas that have
been explored include enum, logging, interface checking, automatic
delegation, automatic property creation, proxies, frameworks, and
automatic resource locking/synchronization.

Here is an example of a metaclass that uses an
"collections.OrderedDict" to remember the order that class variables
are defined:

```
   class OrderedClass(type):

       @classmethod
       def __prepare__(metacls, name, bases, **kwds):
           return collections.OrderedDict()

       def __new__(cls, name, bases, namespace, **kwds):
           result = type.__new__(cls, name, bases, dict(namespace))
           result.members = tuple(namespace)
           return result

   class A(metaclass=OrderedClass):
       def one(self): pass
       def two(self): pass
       def three(self): pass
       def four(self): pass

   >>> A.members
   ('__module__', 'one', 'two', 'three', 'four')
```

When the class definition for *A* gets executed, the process begins
with calling the metaclassâM-^@M-^Ys "__prepare__()" method which returns an
empty "collections.OrderedDict". That mapping records the methods and
attributes of *A* as they are defined within the body of the class
statement. Once those definitions are executed, the ordered dictionary
is fully populated and the metaclassâM-^@M-^Ys "__new__()" method gets
invoked. That method builds the new type and it saves the ordered
dictionary keys in an attribute called "members".

Customizing instance and subclass checks
==

```
The following methods are used to override the default behavior of the
"isinstance()" and "issubclass()" built-in functions.

In particular, the metaclass "abc.ABCMeta" implements these methods in
order to allow the addition of Abstract Base Classes (ABCs) as
âM-^@M-^\virtual base classesâM-^@M-^] to any class or type (including built-in
types), including other ABCs.

class.__instancecheck__(self, instance)

   Return true if *instance* should be considered a (direct or
   indirect) instance of *class*. If defined, called to implement
   "isinstance(instance, class)".

class.__subclasscheck__(self, subclass)

   Return true if *subclass* should be considered a (direct or
   indirect) subclass of *class*.  If defined, called to implement
   "issubclass(subclass, class)".

Note that these methods are looked up on the type (metaclass) of a
class.  They cannot be defined as class methods in the actual class.
This is consistent with the lookup of special methods that are called
on instances, only in this case the instance is itself a class.

See also:

  **PEP 3119** - Introducing Abstract Base Classes
     Includes the specification for customizing "isinstance()" and
     "issubclass()" behavior through "__instancecheck__()" and
     "__subclasscheck__()", with motivation for this functionality in
     the context of adding Abstract Base Classes (see the "abc"
     module) to the language.


Emulating generic types
=======================

One can implement the generic class syntax as specified by **PEP 484**
(for example "List[int]") by defining a special method

classmethod object.__class_getitem__(cls, key)

   Return an object representing the specialization of a generic class
   by type arguments found in *key*.

This method is looked up on the class object itself, and when defined
in the class body, this method is implicitly a class method.  Note,
this mechanism is primarily reserved for use with static type hints,
other usage is discouraged.

See also: **PEP 560** - Core support for typing module and generic
  types


Emulating callable objects
==========================

object.__call__(self[, args...])

   Called when the instance is âM-^@M-^\calledâM-^@M-^] as a function; if this method
   is defined, "x(arg1, arg2, ...)" is a shorthand for
   "x.__call__(arg1, arg2, ...)".


Emulating container types
=========================

The following methods can be defined to implement container objects.
Containers usually are sequences (such as lists or tuples) or mappings
(like dictionaries), but can represent other containers as well.  The
first set of methods is used either to emulate a sequence or to
emulate a mapping; the difference is that for a sequence, the
allowable keys should be the integers *k* for which "0 <= k < N" where
*N* is the length of the sequence, or slice objects, which define a
range of items.  It is also recommended that mappings provide the
methods "keys()", "values()", "items()", "get()", "clear()",
"setdefault()", "pop()", "popitem()", "copy()", and "update()"
behaving similar to those for PythonâM-^@M-^Ys standard dictionary objects.
The "collections.abc" module provides a "MutableMapping" abstract base
class to help create those methods from a base set of "__getitem__()",
"__setitem__()", "__delitem__()", and "keys()". Mutable sequences
```

```
should provide methods "append()", "count()", "index()", "extend()",
"insert()", "pop()", "remove()", "reverse()" and "sort()", like Python
standard list objects.  Finally, sequence types should implement
addition (meaning concatenation) and multiplication (meaning
repetition) by defining the methods "__add__()", "__radd__()",
"__iadd__()", "__mul__()", "__rmul__()" and "__imul__()" described
below; they should not define other numerical operators.  It is
recommended that both mappings and sequences implement the
"__contains__()" method to allow efficient use of the "in" operator;
for mappings, "in" should search the mappingâM-^@M-^Ys keys; for sequences, it
should search through the values.  It is further recommended that both
mappings and sequences implement the "__iter__()" method to allow
efficient iteration through the container; for mappings, "__iter__()"
should be the same as "keys()"; for sequences, it should iterate
through the values.

object.__len__(self)

   Called to implement the built-in function "len()".  Should return
   the length of the object, an integer ">=" 0.  Also, an object that
   doesnâM-^@M-^Yt define a "__bool__()" method and whose "__len__()" method
   returns zero is considered to be false in a Boolean context.

   **CPython implementation detail:** In CPython, the length is
   required to be at most "sys.maxsize". If the length is larger than
   "sys.maxsize" some features (such as "len()") may raise
   "OverflowError".  To prevent raising "OverflowError" by truth value
   testing, an object must define a "__bool__()" method.

object.__length_hint__(self)

   Called to implement "operator.length_hint()". Should return an
   estimated length for the object (which may be greater or less than
   the actual length). The length must be an integer ">=" 0. This
   method is purely an optimization and is never required for
   correctness.

   New in version 3.4.

Note: Slicing is done exclusively with the following three methods.
  A call like

     a[1:2] = b

  is translated to

     a[slice(1, 2, None)] = b

  and so forth.  Missing slice items are always filled in with "None".

object.__getitem__(self, key)

   Called to implement evaluation of "self[key]". For sequence types,
   the accepted keys should be integers and slice objects.  Note that
   the special interpretation of negative indexes (if the class wishes
   to emulate a sequence type) is up to the "__getitem__()" method. If
   *key* is of an inappropriate type, "TypeError" may be raised; if of
   a value outside the set of indexes for the sequence (after any
   special interpretation of negative values), "IndexError" should be
   raised. For mapping types, if *key* is missing (not in the
   container), "KeyError" should be raised.

   Note: "for" loops expect that an "IndexError" will be raised for
     illegal indexes to allow proper detection of the end of the
     sequence.

object.__missing__(self, key)

   Called by "dict"."__getitem__()" to implement "self[key]" for dict
   subclasses when key is not in the dictionary.

object.__setitem__(self, key, value)

   Called to implement assignment to "self[key]".  Same note as for
   "__getitem__()".  This should only be implemented for mappings if
   the objects support changes to the values for keys, or if new keys
   can be added, or for sequences if elements can be replaced.  The
   same exceptions should be raised for improper *key* values as for
   the "__getitem__()" method.

object.__delitem__(self, key)
```

```
   Called to implement deletion of "self[key]".  Same note as for
   "__getitem__()".  This should only be implemented for mappings if
   the objects support removal of keys, or for sequences if elements
   can be removed from the sequence.  The same exceptions should be
   raised for improper *key* values as for the "__getitem__()" method.

object.__iter__(self)

   This method is called when an iterator is required for a container.
   This method should return a new iterator object that can iterate
   over all the objects in the container.  For mappings, it should
   iterate over the keys of the container.

   Iterator objects also need to implement this method; they are
   required to return themselves.  For more information on iterator
   objects, see Iterator Types.

object.__reversed__(self)

   Called (if present) by the "reversed()" built-in to implement
   reverse iteration.  It should return a new iterator object that
   iterates over all the objects in the container in reverse order.

   If the "__reversed__()" method is not provided, the "reversed()"
   built-in will fall back to using the sequence protocol ("__len__()"
   and "__getitem__()").  Objects that support the sequence protocol
   should only provide "__reversed__()" if they can provide an
   implementation that is more efficient than the one provided by
   "reversed()".

The membership test operators ("in" and "not in") are normally
implemented as an iteration through a sequence.  However, container
objects can supply the following special method with a more efficient
implementation, which also does not require the object be a sequence.

object.__contains__(self, item)

   Called to implement membership test operators.  Should return true
   if *item* is in *self*, false otherwise.  For mapping objects, this
   should consider the keys of the mapping rather than the values or
   the key-item pairs.

   For objects that donâM-^@M-^Yt define "__contains__()", the membership test
   first tries iteration via "__iter__()", then the old sequence
   iteration protocol via "__getitem__()", see this section in the
   language reference.


Emulating numeric types
=======================

The following methods can be defined to emulate numeric objects.
Methods corresponding to operations that are not supported by the
particular kind of number implemented (e.g., bitwise operations for
non-integral numbers) should be left undefined.

object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)

   These methods are called to implement the binary arithmetic
   operations ("+", "-", "*", "@", "/", "//", "%", "divmod()",
   "pow()", "**", "<<", ">>", "&", "^", "|").  For instance, to
   evaluate the expression "x + y", where *x* is an instance of a
   class that has an "__add__()" method, "x.__add__(y)" is called.
   The "__divmod__()" method should be the equivalent to using
   "__floordiv__()" and "__mod__()"; it should not be related to
   "__truediv__()".  Note that "__pow__()" should be defined to accept
   an optional third argument if the ternary version of the built-in
   "pow()" function is to be supported.
```

```
   If one of those methods does not support the operation with the
   supplied arguments, it should return "NotImplemented".

object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other)
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)

   These methods are called to implement the binary arithmetic
   operations ("+", "-", "*", "@", "/", "//", "%", "divmod()",
   "pow()", "**", "<<", ">>", "&", "^", "|") with reflected (swapped)
   operands.  These functions are only called if the left operand does
   not support the corresponding operation [3] and the operands are of
   different types. [4] For instance, to evaluate the expression "x -
   y", where *y* is an instance of a class that has an "__rsub__()"
   method, "y.__rsub__(x)" is called if "x.__sub__(y)" returns
   *NotImplemented*.

   Note that ternary "pow()" will not try calling "__rpow__()" (the
   coercion rules would become too complicated).

   Note: If the right operandâM-^@M-^Ys type is a subclass of the left
     operandâM-^@M-^Ys type and that subclass provides the reflected method
     for the operation, this method will be called before the left
     operandâM-^@M-^Ys non-reflected method.  This behavior allows subclasses
     to override their ancestorsâM-^@M-^Y operations.

object.__iadd__(self, other)
object.__isub__(self, other)
object.__imul__(self, other)
object.__imatmul__(self, other)
object.__itruediv__(self, other)
object.__ifloordiv__(self, other)
object.__imod__(self, other)
object.__ipow__(self, other[, modulo])
object.__ilshift__(self, other)
object.__irshift__(self, other)
object.__iand__(self, other)
object.__ixor__(self, other)
object.__ior__(self, other)

   These methods are called to implement the augmented arithmetic
   assignments ("+=", "-=", "*=", "@=", "/=", "//=", "%=", "**=",
   "<<=", ">>=", "&=", "^=", "|=").  These methods should attempt to
   do the operation in-place (modifying *self*) and return the result
   (which could be, but does not have to be, *self*).  If a specific
   method is not defined, the augmented assignment falls back to the
   normal methods.  For instance, if *x* is an instance of a class
   with an "__iadd__()" method, "x += y" is equivalent to "x =
   x.__iadd__(y)" . Otherwise, "x.__add__(y)" and "y.__radd__(x)" are
   considered, as with the evaluation of "x + y". In certain
   situations, augmented assignment can result in unexpected errors
   (see Why does a_tuple[i] += [âM-^@M-^XitemâM-^@M-^Y] raise an exception when the
   addition works?), but this behavior is in fact part of the data
   model.

object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)

   Called to implement the unary arithmetic operations ("-", "+",
   "abs()" and "~").

object.__complex__(self)
object.__int__(self)
object.__float__(self)

   Called to implement the built-in functions "complex()", "int()" and
   "float()".  Should return a value of the appropriate type.

object.__index__(self)
```

```
   Called to implement "operator.index()", and whenever Python needs
   to losslessly convert the numeric object to an integer object (such
   as in slicing, or in the built-in "bin()", "hex()" and "oct()"
   functions). Presence of this method indicates that the numeric
   object is an integer type.  Must return an integer.

   Note: In order to have a coherent integer type class, when
     "__index__()" is defined "__int__()" should also be defined, and
     both should return the same value.

object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)

   Called to implement the built-in function "round()" and "math"
   functions "trunc()", "floor()" and "ceil()". Unless *ndigits* is
   passed to "__round__()" all these methods should return the value
   of the object truncated to an "Integral" (typically an "int").

   If "__int__()" is not defined then the built-in function "int()"
   falls back to "__trunc__()".


With Statement Context Managers
===============================

A *context manager* is an object that defines the runtime context to
be established when executing a "with" statement. The context manager
handles the entry into, and the exit from, the desired runtime context
for the execution of the block of code.  Context managers are normally
invoked using the "with" statement (described in section The with
statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various
kinds of global state, locking and unlocking resources, closing opened
files, etc.

For more information on context managers, see Context Manager Types.

object.__enter__(self)

   Enter the runtime context related to this object. The "with"
   statement will bind this methodâM-^@M-^Ys return value to the target(s)
   specified in the "as" clause of the statement, if any.

object.__exit__(self, exc_type, exc_value, traceback)

   Exit the runtime context related to this object. The parameters
   describe the exception that caused the context to be exited. If the
   context was exited without an exception, all three arguments will
   be "None".

   If an exception is supplied, and the method wishes to suppress the
   exception (i.e., prevent it from being propagated), it should
   return a true value. Otherwise, the exception will be processed
   normally upon exit from this method.

   Note that "__exit__()" methods should not reraise the passed-in
   exception; this is the callerâM-^@M-^Ys responsibility.

See also:

  **PEP 343** - The âM-^@M-^\withâM-^@M-^] statement
     The specification, background, and examples for the Python "with"
     statement.


Special method lookup
=====================

For custom classes, implicit invocations of special methods are only
guaranteed to work correctly if defined on an objectâM-^@M-^Ys type, not in
the objectâM-^@M-^Ys instance dictionary.  That behaviour is the reason why
the following code raises an exception:

   >>> class C:
   ...     pass
   ...
   >>> c = C()
   >>> c.__len__ = lambda: 5
```

```
   >>> len(c)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: object of type 'C' has no len()

The rationale behind this behaviour lies with a number of special
methods such as "__hash__()" and "__repr__()" that are implemented by
all objects, including type objects. If the implicit lookup of these
methods used the conventional lookup process, they would fail when
invoked on the type object itself:

   >>> 1 .__hash__() == hash(1)
   True
   >>> int.__hash__() == hash(int)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: descriptor '__hash__' of 'int' object needs an argument

Incorrectly attempting to invoke an unbound method of a class in this
way is sometimes referred to as âM-^@M-^Xmetaclass confusionâM-^@M-^Y, and is avoided
by bypassing the instance when looking up special methods:

   >>> type(1).__hash__(1) == hash(1)
   True
   >>> type(int).__hash__(int) == hash(int)
   True

In addition to bypassing any instance attributes in the interest of
correctness, implicit special method lookup generally also bypasses
the "__getattribute__()" method even of the objectâM-^@M-^Ys metaclass:

   >>> class Meta(type):
   ...     def __getattribute__(*args):
   ...         print("Metaclass getattribute invoked")
   ...         return type.__getattribute__(*args)
   ...
   >>> class C(object, metaclass=Meta):
   ...     def __len__(self):
   ...         return 10
   ...     def __getattribute__(*args):
   ...         print("Class getattribute invoked")
   ...         return object.__getattribute__(*args)
   ...
   >>> c = C()
   >>> c.__len__()                 # Explicit lookup via instance
   Class getattribute invoked
   10
   >>> type(c).__len__(c)          # Explicit lookup via type
   Metaclass getattribute invoked
   10
   >>> len(c)                      # Implicit lookup
   10

Bypassing the "__getattribute__()" machinery in this fashion provides
significant scope for speed optimisations within the interpreter, at
the cost of some flexibility in the handling of special methods (the
special method *must* be set on the class object itself in order to be
consistently invoked by the interpreter).
```