Torch utility functions

These functions are used in all Torch package for creating and handling classes.
The most interesting function is probably `torch.class()` which allows
the user to create easily new classes. `torch.typename()` might
also be interesting to check what is the class of a given *Torch7* object.

The other functions are for more advanced users.

[metatable] torch.class(name, [parentName], [module])

Creates a new `Torch` class called `name`. If `parentName` is provided, the class will inherit `parentName` methods. A class is a table which has a particular metatable.

If `module` is not provided and if `name` is of the form `package.className` then the class `className` will be added to the specified `package`. In that case, `package` has to be a valid (and already loaded) package. If `name` does not contain any `.`, then the class will be defined in the global environment.

If `module` is provided table, the class will be defined in this table at key `className`.

One [or two] (meta)tables are returned. These tables contain all the method provided by the class [and its parent class if it has been provided]. After a call to `torch.class()` you have to fill-up properly the metatable.

After the class definition is complete, constructing a new class `name` will be achieved by a call to `name()`.
This call will first call the method `lua__init()` if it exists, passing all arguments of `name()`.

```
-- for naming convenience
do
   --- creates a class "Foo"
   local Foo = torch.class('Foo')

   --- the initializer
   function Foo:__init()
```

```
      self.contents = 'this is some text'
   end

   --- a method
   function Foo:print()
      print(self.contents)
   end

   --- another one
   function Foo:bip()
      print('bip')
   end

end

--- now create an instance of Foo
foo = Foo()

--- try it out
foo:print()

--- create a class torch.Bar which
--- inherits from Foo
do
   local Bar, parent = torch.class('torch.Bar', 'Foo')

   --- the initializer
   function Bar:__init(stuff)
      --- call the parent initializer on ourself
      parent.__init(self)

      --- do some stuff
      self.stuff = stuff
   end

   --- a new method
   function Bar:boing()
      print('boing!')
   end

   --- override parent's method
   function Bar:print()
      print(self.contents)
      print(self.stuff)
   end
```

```
end

--- create a new instance and use it
bar = torch.Bar('ha ha!')
bar:print() -- overrided method
bar:boing() -- child method
bar:bip()   -- parent's method
```

For advanced users, it is worth mentionning that `torch.class()` actually calls `torch.newmetatable()` with a particular constructor. The constructor creates a Lua table and set the right metatable on it, and then calls `lua__init()` if it exists in the metatable. It also sets a factory field `lua__factory` such that it is possible to create an empty object of this class.

[string] torch.type(object)

Checks if `object` has a metatable. If it does, and if it corresponds to a `Torch` class, then returns a string containing the name of the class. Otherwise, it returns the Lua `type(object)` of the object. Unlike `torch.typename()`, all outputs are strings:

```
> torch.type(torch.Tensor())
torch.DoubleTensor
> torch.type({})
table
> torch.type(7)
number
```

[string] torch.typename(object)

Checks if `object` has a metatable. If it does, and if it corresponds to a `Torch` class, then returns a string containing the name of the class. Returns `nil` in any other cases.

```
> torch.typename(torch.Tensor())
torch.DoubleTensor
> torch.typename({})
```

```
> torch.typename(7)
```

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

[userdata] torch.typename2id(string)

Given a Torch class name specified by `string`, returns a unique corresponding id (defined by a `lightuserdata` pointing on the internal structure of the class). This might be useful to do a *fast* check of the class of an object (if used with `torch.id()`), avoiding string comparisons.

Returns `nil` if `string` does not specify a Torch object.

[userdata] torch.id(object)

Returns a unique id corresponding to the `class` of the given *Torch7* object. The id is defined by a `lightuserdata` pointing on the internal structure of the class.

Returns `nil` if `object` is not a Torch object.

This is different from the `object` id returned by `torch.pointer()`.

[boolean] isTypeOf(object, typeSpec)

Checks if a given `object` is an instance of the type specified by `typeSpec`. `typeSpec` can be a string (including a `string.find` pattern) or the constructor object for a Torch class. This function traverses up the class hierarchy, so if b is an instance of B which is a subclass of A, then `torch.isTypeOf(b, B)` and `torch.isTypeOf(b, A)` will both return `true`.

[table] torch.newmetatable(name, parentName, constructor)

Register a new metatable as a Torch type with the given string `name`. The new metatable is returned.

If the string `parentName` is not `nil` and is a valid Torch type (previously created by `torch.newmetatable()`) then set the corresponding metatable as a metatable to the returned new
metatable.

If the given `constructor` function is not `nil`, then assign to the variable `name` the given constructor.
The given `name` might be of the form `package.className`, in which case the `className` will be local to the
specified `package`. In that case, `package` must be a valid and already loaded package.

[function] torch.factory(name)

Returns the factory function of the Torch class `name`. If the class name is invalid or if the class has no factory, then returns `nil`.

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

A factory function is able to return a new (empty) object of its corresponding class. This is helpful for
object serialization.

[table] torch.getmetatable(string)

Given a `string`, returns a metatable corresponding to the Torch class described by `string`. Returns `nil` if the class does not exist.

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

Example:

```
> for k, v in pairs(torch.getmetatable('torch.CharStorage')) do
print(k, v) end

__index__       function: 0x1a4ba80
```

```
__typename      torch.CharStorage
write           function: 0x1a49cc0
__tostring__    function: 0x1a586e0
__newindex__    function: 0x1a4ba40
string          function: 0x1a4d860
__version       1
read            function: 0x1a4d840
copy            function: 0x1a49c80
__len__         function: 0x1a37440
fill            function: 0x1a375c0
resize          function: 0x1a37580
__index         table: 0x1a4a080
size            function: 0x1a4ba20
```

[boolean] torch.isequal(object1, object2)

If the two objects given as arguments are *Lua* tables (or *Torch7* objects), then returns `true` if and only if the
tables (or Torch objects) have the same address in memory. Returns `false` in any other cases.

A Torch class is a class created with `torch.class()` or
`torch.newmetatable()`.

[string] torch.getdefaulttensortype()

Returns a string representing the default tensor type currently in use
by *Torch7*.

[table] torch.getenv(function or userdata)

Returns the Lua `table` environment of the given `function` or the given
`userdata`. To know more about environments, please read the documentation
of `lua_setfenv()`
and `lua_getfenv()`.

[number] torch.version(object)

Returns the field `lua__version` of a given object. This might be helpful to handle variations in a class over time.

[number] torch.pointer(object)

Returns a unique id (pointer) of the given `object`, which can be a *Torch7* object, a table, a thread or a function.

This is different from the `class` id returned by `torch.id()`.

torch.setdefaulttensortype([typename])

Sets the default tensor type for all the tensors allocated from this point on. Valid types are:

- `torch.ByteTensor`
- `torch.CharTensor`
- `torch.ShortTensor`
- `torch.IntTensor`
- `torch.FloatTensor`
- `torch.DoubleTensor`

torch.setenv(function or userdata, table)

Assign `table` as the Lua environment of the given `function` or the given `userdata`. To know more about environments, please read the documentation of `lua_setfenv()` and `lua_getfenv()`.

[object] torch.setmetatable(table, classname)

Set the metatable of the given `table` to the metatable of the Torch object named `classname`. This function has to be used with a lot of care.

[table] torch.getconstructortable(string)

BUGGY
Return the constructor table of the Torch class specified by `string`.

[table] torch.totable(object)

Converts a Tensor or a Storage to a lua table. Also available as methods: `tensor:totable()` and `storage:totable()`.
Multidimensional Tensors are converted to a set of nested tables, matching the shape of the source Tensor.

```
> print(torch.totable(torch.Tensor({1, 2, 3})))
{
  1 : 1
  2 : 2
  3 : 3
}
```