

# Geoprocessing with Python

Chris Garrard



MANNING

# *appendix C*

## *OGR*

---

### **C.1 Summary**

#### **DATASOURCE CLASS**

CopyLayer()  
CreateLayer()  
DeleteLayer()  
Dereference()  
Destroy()  
ExecuteSQL()  
GetDriver()  
GetLayer()  
GetLayerByIndex()  
GetLayerByName()  
GetLayerCount()  
GetName()  
GetRefCount()  
GetStyleTable()  
GetSummaryRefCount()  
Reference()  
Release()  
ReleaseResultSet()  
SetStyleTable()  
SyncToDisk()  
TestCapability()  
TestCapability()  
name

#### **DRIVER CLASS**

CopyDataSource()  
CreateDataSource()  
DeleteDataSource()  
Deregister()  
GetName()  
Open()  
Register()  
TestCapability()  
name

#### **FEATURE CLASS**

Clone()  
Dereference()  
Destroy()  
DumpReadable()  
Equal()  
ExportToJson()  
GetDefnRef()  
GetFID()  
GetField()  
GetFieldAsDateTime()  
GetFieldAsDouble()  
GetFieldAsDoubleList()  
GetFieldAsInteger()

GetFieldAsIntegerList()  
 GetFieldAsString()  
 GetFieldAsStringList()  
 GetFieldCount()  
 GetFieldDefnRef()  
 GetFieldIndex()  
 GetFieldType()  
 GetGeomFieldCount()  
 GetGeomFieldDefnRef()  
 GetGeomFieldIndex()  
 GetGeomFieldRef()  
 GetGeometryRef()  
 GetStyleString()  
 IsFieldSet()  
 Reference()  
 SetFID()  
 SetField()  
 SetField2()  
 SetFieldBinaryFromHexString()  
 SetFieldDoubleList()  
 SetFieldIntegerList()  
 SetFieldStringList()  
 SetFrom()  
 SetFromWithMap()  
 SetGeomField()  
 SetGeomFieldDirectly()  
 SetGeometry()  
 SetGeometryDirectly()  
 SetStyleString()  
 UnsetField()  
 geometry()  
 items()  
 keys()

#### **FEATUREDEFN CLASS**

AddFieldDefn()  
 AddGeomFieldDefn()  
 DeleteGeomFieldDefn()  
 Destroy()  
 GetFieldCount()  
 GetFieldDefn()  
 GetFieldIndex()  
 GetGeomFieldCount()

GetGeomFieldDefn()  
 GetGeomFieldIndex()  
 GetGeomType()  
 GetName()  
 GetReferenceCount()  
 IsGeometryIgnored()  
 IsSame()  
 IsStyleIgnored()  
 SetGeomType()  
 SetGeometryIgnored()  
 SetStyleIgnored()  
 SetStyleIgnored()

#### **FIELDDEFN CLASS**

Destroy()  
 GetFieldTypeName()  
 GetJustify()  
 GetName()  
 GetNameRef()  
 GetPrecision()  
 GetType()  
 GetTypeName()  
 GetWidth()  
 IsIgnored()  
 SetIgnored()  
 SetJustify()  
 SetName()  
 SetPrecision()  
 SetType()  
 SetWidth()  
 SetWidth()  
 justify  
 name  
 precision  
 type  
 width

#### **GEOMFIELDDEFN CLASS**

GetName()  
 GetNameRef()  
 GetSpatialRef()  
 GetType()  
 IsIgnored()  
 SetIgnored()

SetName()  
 SetSpatialRef()  
 SetType()  
 SetType()  
 name  
 srs  
 type  
**GEOMETRY CLASS**  
 AddGeometry()  
 AddGeometryDirectly()  
 AddPoint()  
 AddPoint\_2D()  
 Area()  
 AssignSpatialReference()  
 Boundary()  
 Buffer()  
 Centroid()  
 Clone()  
 CloseRings()  
 Contains()  
 ConvexHull()  
 Crosses()  
 Destroy()  
 Difference()  
 Disjoint()  
 Distance()  
 Empty()  
 Equal()  
 Equals()  
 ExportToGML()  
 ExportToJson()  
 ExportToKML()  
 ExportToWkb()  
 ExportToWkt()  
 FlattenTo2D()  
 GetArea()  
 GetBoundary()  
 GetCoordinateDimension()  
 GetDimension()  
 GetEnvelope()  
 GetEnvelope3D()  
 GetGeometryCount()  
 GetGeometryName()

GetGeometryRef()  
 GetGeometryType()  
 GetPoint()  
 GetPointCount()  
 GetPoint\_2D()  
 GetPoints()  
 GetSpatialReference()  
 GetX()  
 GetY()  
 GetZ()  
 Intersect()  
 Intersection()  
 Intersects()  
 IsEmpty()  
 IsRing()  
 IsSimple()  
 IsValid()  
 Length()  
 Overlaps()  
 PointOnSurface()  
 Segmentize()  
 SetCoordinateDimension()  
 SetPoint()  
 SetPoint\_2D()  
 Simplify()  
 SimplifyPreserveTopology()  
 SymDifference()  
 SymmetricDifference()  
 Touches()  
 Transform()  
 TransformTo()  
 Union()  
 UnionCascaded()  
 Within()  
 WkbSize()  
 WkbSize()  
 next()  
**LAYER CLASS**  
 AlterFieldDefn()  
 Clip()  
 CommitTransaction()  
 CreateFeature()  
 CreateField()

CreateFields()  
 CreateGeomField()  
 DeleteFeature()  
 DeleteField()  
 Dereference()  
 Erase()  
 FindFieldIndex()  
 GetExtent()  
 GetFIDColumn()  
 GetFeature()  
 GetFeatureCount()  
 GetFeaturesRead()  
 GetGeomType()  
 GetGeometryColumn()  
 GetLayerDefn()  
 GetName()  
 GetNextFeature()  
 GetRefCount()  
 GetSpatialFilter()  
 GetSpatialRef()  
 GetStyleTable()  
 Identity()  
 Intersection()  
 Reference()  
 ReorderField()  
 ReorderFields()  
 ResetReading()  
 RollbackTransaction()  
 SetAttributeFilter()  
 SetFeature()  
 SetIgnoredFields()  
 SetNextByIndex()  
 SetSpatialFilter()  
 SetSpatialFilterRect()  
 SetStyleTable()  
 StartTransaction()  
 SymDifference()  
 SyncToDisk()  
 TestCapability()  
 Union()  
 Update()

Update()  
 next()  
 schema

#### **STYLETABLE CLASS**

AddStyle()  
 Find()  
 GetLastStyleName()  
 GetNextStyle()  
 LoadStyleTable()  
 ResetStyleStringReading()  
 SaveStyleTable()  
 SaveStyleTable()

#### **MODULE FUNCTIONS**

ApproximateArcAngles()  
 BuildPolygonFromEdges()  
 CreateGeometryFromGML()  
 CreateGeometryFromJson()  
 CreateGeometryFromWkb()  
 CreateGeometryFromWkt()  
 DontUseExceptions()  
 ForceToLineString()  
 ForceToMultiLineString()  
 ForceToMultiPoint()  
 ForceToMultiPolygon()  
 ForceToPolygon()  
 GeneralCmdLineProcessor()  
 GeometryTypeToName()  
 GetDriver()  
 GetDriverByName()  
 GetDriverCount()  
 GetFieldName()  
 GetOpenDS()  
 GetOpenDSCount()  
 GetUseExceptions()  
 Open()  
 OpenShared()  
 RegisterAll()  
 SetGenerate\_DB2\_V72\_BYTE\_ORDER()  
 TermProgress\_nocb()  
 UseExceptions()

## C.2 DataSource class

**CopyLayer(Layer src\_layer, string new\_name, string options = None) -> Layer**

Duplicates an existing layer.

This function creates a new layer, duplicates the field definitions of the source layer, and then duplicates each features of the source layer. The papszOptions argument can be used to control driver-specific creation options. These options are normally documented in the format-specific documentation. The source layer may come from another dataset.

- src\_layer: Handle to the source layer.
- new\_name: The name of the layer to create.
- options: A list of name=value option strings. Options are driver-specific.

Returns a handle to the layer, or None if an error occurs.

**CreateLayer(string name, SpatialReference srs = None, OGRwkbGeometryType geom\_type = wkbUnknown, string options = None) -> Layer**

This function attempts to create a new layer on the data source with the indicated name, coordinate system, geometry type.

The papszOptions argument can be used to control driver specific creation options. These options are normally documented in the format specific documentation.

- name: The name for the new layer. This should ideally not match any existing layer on the data source.
- srs: Handle to the coordinate system to use for the new layer, or None if no coordinate system is available.
- geom\_type: The geometry type for the layer. Use wkbUnknown if no constraints exist on the types geometry to be written.
- options: A list of name=value option strings. Options are driver specific, and driver information can be found at [http://www.gdal.org/ogr/ogr\\_formats.html](http://www.gdal.org/ogr/ogr_formats.html).

None on failure, or a new OGRLayer handle on success.

**DeleteLayer(value)**

Deletes the layer given an index or layer name.

**Dereference()**

For backward compatibility only.

**Destroy()**

Once called, self has effectively been destroyed. Do not access. For backward compatibility only.

**ExecuteSQL(string statement, Geometry spatialFilter = None, string dialect = "") -> Layer**

Executes an SQL statement against the data store.

For more information on the SQL dialect supported internally by OGR review, see the OGR SQL document. Several drivers (that is, Oracle and PostGIS) pass the SQL directly through to the underlying RDBMS.

- **statement:** The SQL statement to execute.
- **spatialFilter:** Handle to a geometry which represents a spatial filter. Can be None.
- **dialect:** Allows control of the statement dialect. If set to None, the OGR SQL engine will be used, except for RDBMS drivers that will use their dedicated SQL engine, unless OGRSQL is explicitly passed as the dialect.

Returns a handle to an OGRLayer containing the results of the query. Deallocate with OGR\_DS\_ReleaseResultSet().

**GetDriver() -> Driver**

Returns the driver that the dataset was opened with.

Returns None if driver info isn't available, or pointer to a driver owned by the OGRSFDriverManager.

**GetLayer(iLayer=0)**

Returns the layer given an index or a name.

**GetLayerByIndex(int index = 0) -> Layer**

**GetLayerByName(string layer\_name) -> Layer**

Fetches a layer by name.

The returned layer remains owned by the OGRDataSource and shouldn't be deleted by the application.

- **layer\_name:** The layer name of the layer to fetch.

Returns a handle to the layer, or None if the layer isn't found or an error occurs.

**GetLayerCount() -> int**

Gets the number of layers in this data source.

Returns layer count.

**GetName() -> char**

Returns the name of the data source.

This string should be sufficient to open the data source if passed to the same OGRSFDriver that this data source was opened with, but it need not be exactly the same string that was used to open the data source. Normally this is a filename.

Returns pointer to an internal name string that shouldn't be modified or freed by the caller.

**GetRefCount() -> int**

**GetStyleTable() -> StyleTable**

**GetSummaryRefCount() -> int**

**Reference()**

For backward compatibility only.

**Release()**

Once called, self has effectively been destroyed. Do not access. For backward compatibility only.

**ReleaseResultSet(Layer layer)**

Releases results of OGR\_DS\_ExecuteSQL().

This function should only be used to deallocate OGRLayers resulting from an OGR\_DS\_ExecuteSQL() call on the same OGRDataSource. Failure to deallocate a results set before destroying the OGRDataSource may cause errors.

- layer: Handle to the result of a previous OGR\_DS\_ExecuteSQL() call

**SetStyleTable(StyleTable table)**

**SyncToDisk() -> OGRERR**

Flushes pending changes to disk.

This call is intended to force the data source to flush any pending writes to disk, and leave the disk file in a consistent state. It wouldn't normally have any effect on read-only data sources.

Several data sources do not implement this method, and will still return OGRERR\_NONE. An error is only returned if an error occurs while attempting to flush to disk.

The default implementation of this method calls the SyncToDisk() method on each of the layers. Conceptionally, calling SyncToDisk() on a data source should include any work that might be accomplished by calling SyncToDisk() on layers in that data source.

Returns OGRERR\_NONE if no error occurs (even if nothing is done) or an error code.

**TestCapability(string cap) -> bool**

Tests if capability is available.



One of the following data source capability names can be passed into this function, and a True or False value will be returned indicating whether or not the capability is available for this object.

ODsCCreateLayer: True if this data source can create new layers

The #define macro forms of the capability names should be used in preference to the strings themselves to avoid misspelling.

- cap: The capability to test

Returns True if capability available otherwise False.

### **TestCapability(string cap) -> bool**

Tests if capability is available.

One of the following data source capability names can be passed into this function, and a True or False value will be returned indicating whether or not the capability is available for this object.

ODsCCreateLayer: True if this data source can create new layers

The #define macro forms of the capability names should be used in preference to the strings themselves to avoid misspelling.

- cap: The capability to test.

Returns True if capability available otherwise False.

### **PROPERTIES**

**name**

DataSource\_name\_get(DataSource self) -> char

## **C.3 Driver class**

**CopyDataSource(DataSource copy\_ds, string utf8\_path, string options = None) -> DataSource**

Creates a new data source by copying all the layers from the source data source.

It's important to call OGR\_DS\_Destroy() when the data source is no longer used to ensure that all data has been properly flushed to disk.

- copy\_ds: Source data source.
- utf8\_path: The name for the new data source.
- options: A list of name=value option strings. Options are driver specific, and driver information can be found at [http://www.gdal.org/ogr/ogr\\_formats.html](http://www.gdal.org/ogr/ogr_formats.html).

Returns None is returned on failure, or a new OGRDataSource handle on success.

**CreateDataSource(string utf8\_path, string options = None) -> DataSource**

This function attempts to create a new data source based on the passed driver.

The papszOptions argument can be used to control driver specific creation options. These options are normally documented in the format specific documentation.

It's important to call `OGR_DS_Destroy()` when the data source is no longer used to ensure that all data has been properly flushed to disk.

- `utf8_path`: The name for the new data source. UTF-8 encoded.
- `options`: A list of name=value option strings. Options are driver specific, and driver information can be found at [http://www.gdal.org/ogr/ogr\\_formats.html](http://www.gdal.org/ogr/ogr_formats.html).

Returns `None` is returned on failure, or a new `OGRDataSource` handle on success.

### **DeleteDataSource(string utf8\_path) -> int**

Deletes a data source.

Delete (from the disk, in the database, ...) the named data source. Normally it would be safest if the data source wasn't open at the time.

Whether this is a supported operation on this driver case be tested using `TestCapability()` on `ODrCDeleteDataSource`.

- `utf8_path`: The name of the data source to delete.

Returns `OGRERR_NONE` on success, and `OGRERR_UNSUPPORTED_OPERATION` if this isn't supported by this driver.

### **Deregister()**

### **GetName() -> char**

Fetches name of driver (file format). This name should be relatively short (10-40 characters), and should reflect the underlying file format. For instance, "ESRI Shapefile".

Returns driver name. This is an internal string and shouldn't be modified or freed.

### **Open(string utf8\_path, int update = 0) -> DataSource**

Attempts to open file with this driver.

- `utf8_path`: The name of the file, or data source to try and open.
- `update`: True if update access is required; otherwise, False (the default).

Returns `None` on error or if the pass name isn't supported by this driver; otherwise, a handle to an `OGRDataSource`. This `OGRDataSource` should be closed by deleting the object when it's no longer needed.

### **Register()**

### **TestCapability(string cap) -> bool**

Tests if capability is available.

One of the following data source capability names can be passed into this function, and a True or False value will be returned indicating whether or not the capability is available for this object.

ODrCCreateDataSource: True if this driver can support creating data sources.

ODrCDeleteDataSource: True if this driver supports deleting data sources.

The #define macro forms of the capability names should be used in preference to the strings themselves to avoid misspelling.

- cap: The capability to test

Returns True if capability available otherwise False.

#### **PROPERTIES**

**name**

Driver\_name\_get(Driver self) -> char

## **C.4 Feature class**

**Clone()** -> Feature

Duplicates feature.

The newly created feature is owned by the caller, and will have its own reference to the OGRFeatureDefn.

Returns a handle to the new feature, exactly matching this feature.

**Dereference()**

**Destroy()**

Once called, self has effectively been destroyed. Do not access. For backward compatibility only.

**DumpReadable()**

Dumps this feature in a human readable form.

This dumps the attributes, and geometry; however, it doesn't definition information (other than field types and names), nor does it report the geometry spatial reference system.

- fpOut: The stream to write to, such as strout

**Equal(Feature feature)** -> bool

Tests if two features are the same.

Two features are considered equal if they share them (handle equality) same OGRFeatureDefn, have the same field values, and the same geometry (as tested by OGR\_G\_Equal()) as well as the same feature id.

- feature: Handle to the other feature to test this one against

Returns True if they are equal; otherwise, False.

**ExportToJson(as\_object=False, options=None)**

Exports a GeoJSON object which represents the Feature. The as\_object parameter determines whether the returned value should be a Python object instead of a string. Defaults to False. The options parameter is passed to Geometry.ExportToJson()

**geometry() -> Geometry**

Returns the feature's geometry.

**GetDefnRef() -> FeatureDefn**

Fetches feature definition.

Returns a handle to the feature definition object on which feature depends.

**GetFID() -> int**

Gets feature identifier.

Returns feature id or OGRNullFID if none has been assigned.

**GetField(fld\_index)****GetFieldAsDateTime(int id)**

Fetches field value as date and time.

Currently this method only works for OFTDate, OFTTime, and OFTDateTime fields.

- id: The field to fetch, from 0 to GetFieldCount()-1
- pnYear: (including century)
- pnMonth: (1-12)
- pnDay: (1-31)
- pnHour: (0-23)
- pnMinute: (0-59)
- pnSecond: (0-59)
- pnTZFlag: (0=unknown, 1=localtime, 100=GMT, see data model for details)

Returns True on success or False on failure.

**GetFieldAsDouble(string name) -> double**

Fetches field value as a double.

OFTString features will be translated using atof(). OFTInteger fields will be cast to double. Other field types, or errors will result in a return value of zero.

- name: The field to fetch, from 0 to GetFieldCount()-1

Returns the field value.

**GetFieldAsDoubleList(int id)**

Fetches field value as a list of doubles.

Currently this function only works for OFTRealList fields.

- id: The field to fetch, from 0 to GetFieldCount()-1
- pnCount: An integer to put the list count (number of doubles) into

Returns the field value. This list is internal, and shouldn't be modified, or freed. Its lifetime may be brief. If \*pnCount is zero on return, the returned pointer may be None or non-None.

**GetFieldAsInteger(string name) -> int**

Fetches field value as integer.

OFTString features will be translated using atoi(). OFTReal fields will be cast to integer. Other field types or errors will result in a return value of zero.

- name: The field to fetch, from 0 to GetFieldCount()-1

Returns the field value.

**GetFieldAsIntegerList(int id)**

Fetches field value as a list of integers.

Currently this function only works for OFTIntegerList fields.

- id: The field to fetch, from 0 to GetFieldCount()-1
- pnCount: An integer to put the list count (number of integers) into

Returns the field value. This list is internal, and shouldn't be modified, or freed. Its lifetime may be brief. If \*pnCount is zero on return, the returned pointer may be None or non-None.

**GetFieldAsString(string name) -> char**

Fetches field value as a string.

- name: The field to fetch, from 0 to GetFieldCount()-1

Returns the field value. This string is internal, and shouldn't be modified, or freed. Its lifetime may be brief.

**GetFieldAsStringList(int id) -> char**

Fetches field value as a list of strings.

Currently this method only works for OFTStringList fields.

The returned list is terminated by a None pointer. The number of elements can also be calculated using CSLCount().

- id: The field to fetch, from 0 to GetFieldCount()-1

Returns the field value. This list is internal, and shouldn't be modified, or freed. Its lifetime may be brief.

**GetFieldCount() -> int**

Fetches number of fields on this feature. This will always be the same as the field count for the OGRFeatureDefn.

Returns count of fields.

**GetFieldDefnRef(string name) -> FieldDefn**

Fetches definition for this field.

- name: The field to fetch, from 0 to GetFieldCount()-1

Returns a handle to the field definition (from the OGRFeatureDefn). This is an internal reference, and shouldn't be deleted or modified.

**GetFieldIndex(string name) -> int**

Fetches the field index given field name.

This is a cover for the OGRFeatureDefn::GetFieldIndex() method.

- name: The name of the field to search for

Returns the field index, or -1 if no matching field is found.

**GetFieldType(string name) -> OGRFieldType**

**GetGeomFieldCount() -> int**

**GetGeomFieldDefnRef(string name) -> GeomFieldDefn**

**GetGeomFieldIndex(string name) -> int**

**GetGeomFieldRef(string name) -> Geometry**

**GetGeometryRef() -> Geometry**

Fetches a handle to feature geometry.

Returns a handle to internal feature geometry. This object shouldn't be modified.

**GetStyleString() -> char**

Fetches style string for this feature.

Set the OGR Feature Style Specification for details on the format of this string, and ogr\_featurestyle.h for services available to parse it.

Returns a reference to a representation in string format, or None if there isn't one.

**IsFieldSet(string name) -> bool**

Tests if a field has ever been assigned a value or not.

- name: The field to test

Returns True if the field has been set; otherwise, false.

**items() -> dictionary**

Returns a dictionary containing the feature's attribute values, with field names as keys.

**keys() -> list**

Returns a list of attribute field names.

**Reference()**

**SetFID(int fid) -> OGRErr**

Sets the feature identifier.

For specific types of features this operation may fail on illegal features ids. Generally, it always succeeds. Feature ids should be greater than or equal to zero, with the exception of OGRNullFID (-1) indicating that the feature id is unknown.

- **fid:** The new feature identifier value to assign

Returns On success OGRERR\_NONE, or on failure of another value.

**SetField(string name, int year, int month, int day, int hour, int minute, int second, int tzflag)**

**SetField2(fld\_index, value)**

**SetFieldBinaryFromHexString(string name, string pszValue)**

**SetFieldDoubleList(int id, int nList)**

Sets field to list of doubles value.

This function currently on has an effect of OFTRealList fields.

- **id:** The field to set, from 0 to GetFieldCount()-1
- **nList:** The number of values in the list being assigned
- **padfValues:** The values to assign

**SetFieldIntegerList(int id, int nList)**

Sets field to list of integers value.

This function currently on has an effect of OFTIntegerList fields.

- **id:** The field to set, from 0 to GetFieldCount()-1
- **nList:** The number of values in the list being assigned
- **panValues:** The values to assign

**SetFieldStringList(int id, string pList)**

Sets field to list of strings value.

This function currently on has an effect of OFTStringList fields.

- **id:** The field to set, from 0 to GetFieldCount()-1
- **pList:** The values to assign

**SetFrom(Feature other, int forgiving = 1) -> OGRErr**

Sets one feature from another.

Overwrite the contents of this feature from the geometry and attributes of another. The hOtherFeature doesn't need to have the same OGRFeatureDefn. Field values are copied by corresponding field names. Field types don't have to exactly match. OGR\_F\_SetField\*() function conversion rules will be applied as needed.

- **other:** Handle to the feature from which geometry, and field values, will be copied
- **forgiving:** True if the operation should continue despite lacking output fields matching some of the source fields

Returns OGRERR\_NONE if the operation succeeds, even if several values aren't transferred; otherwise, an error code.

### **SetFromWithMap(*Feature other*, *int forgiving*, *int nList*) -> OGRERR**

Sets one feature from another.

Overwrite the contents of this feature from the geometry and attributes of another. The *hOtherFeature* doesn't need to have the same OGRFeatureDefn. Field values are copied according to the provided indices map. Field types don't have to exactly match. OGR\_F\_SetField\*() function conversion rules will be applied as needed. This is more efficient than OGR\_F\_SetFrom() in that this doesn't look up the fields by their names. Particularly useful when the field names don't match.

- *other*: Handle to the feature from which geometry, and field values will be copied.
- *forgiving*: Array of the indices of the destination feature's fields stored at the corresponding index of the source feature's fields. A value of -1 should be used to ignore the source's field. The array shouldn't be None and be as long as the number of fields in the source feature.
- *nList*: True if the operation should continue despite lacking output fields matching several of the source fields.

Returns OGRERR\_NONE if the operation succeeds, even if several values aren't transferred; otherwise, an error code.

### **SetGeomField(*string name*, *Geometry geom*) -> OGRERR**

### **SetGeomFieldDirectly(*string name*, *Geometry geom*) -> OGRERR**

### **SetGeometry(*Geometry geom*) -> OGRERR**

Sets feature geometry.

- *geom*: Handle to the new geometry to apply to feature

Returns OGRERR\_NONE if successful, or OGR\_UNSUPPORTED\_GEOMETRY\_TYPE if the geometry type is illegal for the OGRFeatureDefn (checking not yet implemented).

### **SetGeometryDirectly(*Geometry geom*) -> OGRERR**

Sets feature geometry.

- *geom*: Handle to the new geometry to apply to feature

Returns OGRERR\_NONE if successful, or OGR\_UNSUPPORTED\_GEOMETRY\_TYPE if the geometry type is illegal for the OGRFeatureDefn (checking not yet implemented).

### **SetStyleString(*string the\_string*)**

- *the\_string*: The style string to apply to this feature, cannot be None



**UnsetField(string name)**

Clears a field, marking it as unset.

- name: The field to unset

**C.5 *FeatureDefn class*****AddFieldDefn(FieldDefn defn)**

Adds a new field definition to the passed feature definition.

To add a new field definition to a layer definition, don't use this function directly, but use `OGR_L_CreateField()` instead.

This function should only be called while there are no `OGRFeature` objects in existence based on this `OGRFeatureDefn`. The `OGRFieldDefn` passed in is copied, and remains the responsibility of the caller.

- defn: Handle to the new field definition

**AddGeomFieldDefn(GeomFieldDefn defn)****DeleteGeomFieldDefn(int idx) -> OGRErr****Destroy()**

Once called, self has effectively been destroyed. Don't access. For backward compatibility only.

**GetFieldCount() -> int**

Fetches number of fields on the passed feature definition.

Returns count of fields.

**GetFieldDefn(int i) -> FieldDefn**

Fetches field definition of the passed feature definition.

Starting with GDAL 1.7.0, this method will also issue an error if the index isn't valid.

- i: The field to fetch, between 0 and `GetFieldCount()-1`

Returns a handle to an internal field definition object or `None` if invalid index. This object shouldn't be modified or freed by the application.

**GetFieldIndex(string name) -> int**

Finds field by name.

The field index of the first field matching the passed field name (case insensitively) is returned.

- name: The field name to search for

Returns the field index, or -1 if no match found.

**GetGeomFieldCount() -> int**

**GetGeomFieldDefn(int i) -> GeomFieldDefn**

**GetGeomFieldIndex(string name) -> int**

**GetGeomType() -> OGRwkbGeometryType**

Fetches the geometry base type of the passed feature definition.

Returns the base type for all geometry related to this definition.

**GetName() -> char**

Gets name of the OGRFeatureDefn passed as an argument.

Returns the name. This name is internal and shouldn't be modified or freed.

**GetReferenceCount() -> int**

Fetches current reference count.

Returns the current reference count.

**IsGeometryIgnored() -> int**

Determines whether the geometry can be omitted when fetching features.

Returns ignore state.

**IsSame(FeatureDefn other\_defn) -> int**

**IsStyleIgnored() -> int**

Determines whether the style can be omitted when fetching features.

Returns ignore state.

**SetGeomType(OGRwkbGeometryType geom\_type)**

Assigns the base geometry type for the passed layer (the same as the feature definition).

All geometry objects using this type must be of the defined type or a derived type. The default upon creation is wkbUnknown, which allows for any geometry type. The geometry type should generally not be changed after any OGRFeatures have been created against this definition.

- geom\_type: The new type to assign

**SetGeometryIgnored(int bIgnored)**

Sets whether the geometry can be omitted when fetching features.

- bIgnored: Ignore state

**SetStyleIgnored(int bIgnored)**

Sets whether the style can be omitted when fetching features.

- bIgnored: Ignore state

**SetStyleIgnored(int bIgnored)**

Sets whether the style can be omitted when fetching features.

- bIgnored: Ignore state

**C.6 FieldDefn class****Destroy()**

Once called, self has effectively been destroyed. Don't access. For backward compatibility only.

**GetFieldTypeName(OGRFieldType type) -> char****GetJustify() -> OGRJustification**

Gets the justification for this field.

Returns the justification.

**GetName() -> char****GetNameRef() -> char**

Fetches name of this field.

Returns the name of the field definition.

**GetPrecision() -> int**

Gets the formatting precision for this field. This should normally be zero for fields of types other than OFTReal.

Returns the precision.

**GetType() -> OGRFieldType**

Fetches type of this field.

Returns field type.

**GetTypeNames() -> char****GetWidth() -> int**

Gets the formatting width for this field.

Returns the width; zero means no specified width.

**IsIgnored() -> int**

Returns whether this field should be omitted when fetching features.

Returns ignore state.

**SetIgnored(int bIgnored)**

Sets whether this field should be omitted when fetching features.

- bIgnored: Ignore state

**SetJustify(OGRJustification justify)**

Sets the justification for this field.

- justify: The new justification

**SetName(string name)**

Resets the name of this field.

- name: The new name to apply

**SetPrecision(int precision)**

Sets the formatting precision for this field in characters.

This should normally be zero for fields of types other than OFTReal.

- precision: The new precision

**SetType(OGRFieldType type)**

Sets the type of this field. This should never be done to an OGRFieldDefn that is already part of an OGRFeatureDefn.

- type: The new field type

**SetWidth(int width)**

Sets the formatting width for this field in characters.

- width: The new width

**GetWidth(int width)**

Sets the formatting width for this field in characters.

- width: The new width

**PROPERTIES****justify**

Gets the justification for this field.

**name**

GetName(self) -> char

**precision**

Gets the formatting precision for this field. This should normally be zero for fields of types other than OFTReal.

**type**

Fetches type of this field.

**width**

Gets the formatting width for this field.

## **C.7 *GeomFieldDefn* class**

`GetName()` -> char

`GetNameRef()` -> char

`GetSpatialRef()` -> SpatialReference

`GetType()` -> OGRwkbGeometryType

`IsIgnored()` -> int

`SetIgnored(int bIgnored)`

`SetName(string name)`

`SetSpatialRef(SpatialReference srs)`

`SetType(OGRwkbGeometryType type)`

`SetType(OGRwkbGeometryType type)`

### **PROPERTIES**

**name**

`GetName(self)` -> char

**srs**

`GetSpatialRef(self)` -> SpatialReference

**type**

`GetType(self)` -> OGRwkbGeometryType

## **C.8 *Geometry* class**

`AddGeometry(Geometry other)` -> OGRErr

`AddGeometryDirectly(Geometry other_disown)` -> OGRErr

`AddPoint(double x, double y, double z = 0)`

`AddPoint_2D(double x, double y)`

`Area()` -> double

**AssignSpatialReference(SpatialReference reference)**

Assigns spatial reference to this object.

Any existing spatial reference is replaced, but under no circumstances does this result in the object being reprojected. It's changing the interpretation of the existing geometry. Note that assigning a spatial reference increments the reference count on the OGRSpatialReference, but doesn't copy it.

This is similar to the SFCOM IGeometry::put\_SpatialReference() method.

- reference: Handle on the new spatial reference system to apply

**Boundary() -> Geometry**

Computes boundary.

A new geometry object is created and returned containing the boundary of the geometry on which the method is invoked.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- A handle to a newly allocated geometry now owned by the caller or None on failure

Returns OGR 1.8.0

**Buffer(double distance, int quadsecs = 30) -> Geometry**

Computes buffer of geometry.

Builds a new geometry containing the buffer region around the geometry on which it's invoked. The buffer is a polygon containing the region within the buffer distance of the original geometry.

Certain buffer sections are properly described as curves, but are converted to approximate polygons. The nQuadSegs parameter can be used to control how many segments should be used to define a 90-degree curve—a quadrant of a circle. A value of 30 is a reasonable default. Large values result in large numbers of vertices in the resulting buffer geometry, while small numbers reduce the accuracy of the result.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- distance: The buffer distance to be applied
- quadsecs: The number of segments used to approximate a 90 degree (quadrant) of curvature

Returns the newly created geometry, or None if an error occurs.

**Centroid() -> Geometry**

Computes the geometry centroid.

The centroid location is applied to the passed in OGRPoint object. The centroid isn't necessarily within the geometry.

This method relates to the SFCOM ISurface::get\_Centroid() method; however, the current implementation based on GEOS can operate on other geometry types such as multipoint and linestring, and geometrycollections such as multipolygons. OGC SQL 1.1 defines the operation for surfaces (polygons). SQL/MM-Part 3 defines the operation for surfaces and multisurfaces (multipolygons).

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

Returns OGRERR\_NONE on success or OGRERR\_FAILURE on error.

### **Clone() -> Geometry**

Make a copy of this object.

This function relates to the SFCOM IGeometry::clone() method.

Returns a handle on the copy of the geometry with the spatial reference system as the original.

### **CloseRings()**

Forces rings to be closed.

If this geometry, or any contained geometries has polygon rings that aren't closed, they will be closed by adding the starting point at the end.

### **Contains(Geometry other) -> bool**

Tests for containment.

Tests if this geometry contains the other geometry.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare

Returns True if hThis contains hOther geometry; otherwise, False.

### **ConvexHull() -> Geometry**

Computes convex hull.

A new geometry object is created and returned containing the convex hull of the geometry on which the method is invoked.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

Returns a handle to a newly allocated geometry now owned by the caller or None on failure.

**Crosses(Geometry other) -> bool**

Tests for crossing.

Tests if this geometry and the other geometry are crossing.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare

Returns True if they are crossing; otherwise, False.

**Destroy()****Difference(Geometry other) -> Geometry**

Computes difference.

Generates a new geometry which is the region of this geometry with the region of the other geometry removed.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry

Returns a new geometry representing the difference or None if the difference is empty or an error occurs.

**Disjoint(Geometry other) -> bool**

Tests for disjointedness.

Tests if this geometry and the other geometry are disjoint.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare

Returns True if they are disjoint; otherwise, False.

**Distance(Geometry other) -> double**

Computes distance between two geometries.

Returns the shortest distance between the two geometries.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare against

Returns the distance between the geometries or -1 if an error occurs.



**Empty()**

Clears geometry information. This restores the geometry to its initial state after construction and before assignment of actual geometry.

This function relates to the SFCOM IGeometry::Empty() method.

**Equal(Geometry other) -> bool****Equals(Geometry other) -> bool**

Returns True if two geometries are equivalent.

- other: Handle on the other geometry to test against

Returns True if equivalent or False otherwise.

**ExportToGML(string options = None) -> retStringAndCPLFree****ExportToJson(string options = None) -> retStringAndCPLFree****ExportToKML(string altitude\_mode = None) -> retStringAndCPLFree****ExportToWkb(OGRwkbByteOrder byte\_order = wkbXDR) -> OGRErr**

Converts a geometry into well-known binary format.

This function relates to the SFCOM IWks::ExportToWKB() method.

- byte\_order: One of wkbXDR or wkbNDR indicating MSB or LSB byte order, respectively.
- pabyDstBuffer: A buffer into which the binary representation is written. This buffer must be at least OGR\_G\_WkbSize() byte in size.

Returns Currently OGRERR\_NONE is always returned.

**ExportToWkt() -> OGRErr**

Converts a geometry into well-known text format.

This function relates to the SFCOM IWks::ExportToWKT() method.

- ppszSrcText: A text buffer is allocated by the program, and assigned to the passed pointer.

Returns Currently OGRERR\_NONE is always returned.

**FlattenTo2D()**

Converts geometry to strictly 2D. In a sense this converts all Z coordinates to 0.0.

**GetArea() -> double****GetBoundary() -> Geometry**

Computes boundary (deprecated).

Returns Deprecated See: `OGR_G_Boundary()`.

**GetCoordinateDimension() -> int**

Gets the dimension of the coordinates in this geometry.

This function corresponds to the `SFCOM IGeometry::GetDimension()` method.

Returns in practice this will return 2 or 3. It can also return 0 in the case of an empty point.

**GetDimension() -> int**

Gets the dimension of this geometry.

Returns 0 for points, 1 for lines, and 2 for surfaces.

**GetEnvelope()**

Computes and returns the bounding envelope for this geometry in the passed `psEnvelope` structure.

- `psEnvelope`: The structure in which to place the results

**GetEnvelope3D()**

Computes and returns the bounding envelope (3D) for this geometry in the passed `psEnvelope` structure.

- `psEnvelope`: The structure in which to place the results

Returns OGR 1.9.0

**GetGeometryCount() -> int****GetGeometryName() -> char**

Fetches WKT name for geometry type.

There's no SFCOM analog to this function.

Returns name used for this geometry type in well-known text format.

**GetGeometryRef(int geom) -> Geometry****GetGeometryType() -> OGRwkbGeometryType**

Fetches geometry type.

Note that the geometry type may include the 2.5D flag. To get a 2D flattened version of the geometry type, apply the `wkbFlatten()` macro to the return result.

Returns the geometry type code.

**GetPoint(int iPoint = 0)****GetPointCount() -> int**

**GetPoint\_2D(int iPoint = 0)**

**GetPoints(int nCoordDimension = 0)**

**GetSpatialReference() -> SpatialReference**

Returns spatial reference system for geometry.

This function relates to the SFCOM IGeometry::get\_SpatialReference() method.

Returns a reference to the spatial reference geometry.

**GetX(int point = 0) -> double**

**GetY(int point = 0) -> double**

**GetZ(int point = 0) -> double**

**Intersect(Geometry other) -> bool**

**Intersection(Geometry other) -> Geometry**

Computes intersection.

Generates a new geometry that's the region of intersection of the two geometries operated on. The OGR\_G\_Intersects() function can be used to test if two geometries intersect.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry

Returns a new geometry representing the intersection or None if there's no intersection or an error occurs.

**Intersects(Geometry other) -> bool**

Do these features intersect?

Currently this is not implemented in a rigorous fashion, and generally tests whether the envelopes of the two features intersect. Eventually this will be made rigorous.

- other: Handle on the other geometry to test against

Returns True if the geometries intersect; otherwise, False.

**IsEmpty() -> bool**

Tests if the geometry is empty.

Returns True if the geometry has no points; otherwise, False.

**IsRing() -> bool**

Test if the geometry is a ring.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always return False.

Returns True if the geometry has no points; otherwise, False.

**IsSimple() -> bool**

Returns True if the geometry is simple.

Returns True if the geometry has no anomalous geometric points, such as self-intersection or self-tangency. The description of each instantiable geometric class will include the specific conditions that cause an instance of that class to be classified as not simple.

If OGR is built without the GEOS library, this function will always return False.

Returns True if object is simple; otherwise, False.

**IsValid() -> bool**

Tests if the geometry is valid.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always return False.

Returns True if the geometry has no points; otherwise, False.

**Length() -> double****Overlaps(Geometry other) -> bool**

Tests for overlap.

Tests if this geometry and the other geometry overlap; that is, their intersection has a non-zero area.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare

Returns True if they are overlapping; otherwise, False.

**PointOnSurface() -> Geometry****Segmentize(double dfMaxLength)**

Modifies the geometry such that it has no segment longer than the given distance. Interpolated points will have Z and M values (if needed) set to 0. Distance computation is performed in 2D only.

- dfMaxLength: The maximum distance between 2 points after segmentization

**SetCoordinateDimension(int dimension)**

Sets the coordinate dimension.

This method sets the explicit coordinate dimension. Setting the coordinate dimension of a geometry to 2 should zero-out any existing Z values. Setting the dimension of a geometry collection will not necessarily affect the children geometries.

- dimension: New coordinate dimension value, either 2 or 3

**SetPoint(int point, double x, double y, double z = 0)****SetPoint\_2D(int point, double x, double y)****Simplify(double tolerance) -> Geometry**

Computes a simplified geometry.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- tolerance: The distance tolerance for the simplification
- The simplified geometry or None if an error occurs

Returns OGR 1.8.0.

**SimplifyPreserveTopology(double tolerance) -> Geometry**

Computes a simplified geometry.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- tolerance: The distance tolerance for the simplification
- The simplified geometry or None if an error occurs

Returns OGR 1.9.0.

**SymDifference(Geometry other) -> Geometry**

Computes symmetric difference.

Generates a new geometry that is the symmetric difference of this geometry and the other geometry.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry
- A new geometry representing the symmetric difference or None if the difference is empty or an error occurs.

Returns OGR 1.8.0.

**SymmetricDifference(Geometry other) -> Geometry**

Compute symmetric difference (deprecated).

Returns Deprecated See: `OGR_G_SymmetricDifference()`.

**Touches(Geometry other) -> bool**

Tests for touching.

Tests if this geometry and the other geometry are touching.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a `CPLE_NotSupported` error.

- other: The other geometry to compare

Returns True if they are touching; otherwise, False.

**Transform(CoordinateTransformation trans) -> OGRErr**

Applies arbitrary coordinate transformation to geometry.

This function will transform the coordinates of a geometry from their current spatial reference system to a new target spatial reference system. Normally this means reprojecting the vectors, but it could include datum shifts and changes of units.

Note that this function does not require that the geometry already have a spatial reference system. It will be assumed that it can be treated as having the source spatial reference system of the `OGRCoordinateTransformation` object, and the actual SRS of the geometry will be ignored. On successful completion, the output `OGRSpatialReference` of the `OGRCoordinateTransformation` will be assigned to the geometry.

- trans: Handle on the transformation to apply

Returns `OGRERR_NONE` on success or an error code.

**TransformTo(SpatialReference reference) -> OGRErr**

Transforms geometry to new spatial reference system.

This function will transform the coordinates of a geometry from their current spatial reference system to a new target spatial reference system. Normally this means reprojecting the vectors, but it could include datum shifts and changes of units.

This function will only work if the geometry already has an assigned spatial reference system, and if it is transformable to the target coordinate system.

- reference: Handle on the spatial reference system to apply

Returns `OGRERR_NONE` on success, or an error code.

**Union(Geometry other) -> Geometry**

Computes union.

Generates a new geometry that is the region of union of the two geometries operated on.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry

Returns a new geometry representing the union or None if an error occurs.

#### **UnionCascaded() -> Geometry**

Computes union using cascading.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

Returns a new geometry representing the union or None if an error occurs.

#### **Within(Geometry other) -> bool**

Tests for containment.

Tests if this geometry is within the other geometry.

This function is built on the GEOS library; check it for the definition of the geometry operation. If OGR is built without the GEOS library, this function will always fail, issuing a CPLE\_NotSupported error.

- other: The other geometry to compare

Returns True if hThis is within hOther; otherwise, False.

#### **WkbSize() -> int**

Returns size of related binary representation.

This function returns the exact number of bytes required to hold the well-known binary representation of this geometry object. Its computation may be slightly expensive for complex geometries.

This function relates to the SFCOM IWks::WkbSize() method.

Returns size of binary representation in bytes.

#### **WkbSize() -> int**

Returns size of related binary representation.

This function returns the exact number of bytes required to hold the well-known binary representation of this geometry object. Its computation may be slightly expensive for complex geometries.

This function relates to the SFCOM IWks::WkbSize() method.

Returns size of binary representation in bytes.

#### **next()**

## C.9 Layer class

**AlterFieldDefn(int iField, FieldDefn field\_def, int nFlags) -> OGRErr**

Alters the definition of an existing field on a layer.

You must use this to alter the definition of an existing field of a real layer. Internally the OGRFeatureDefn for the layer will be updated to reflect the altered field. Applications should never modify the OGRFeatureDefn used by a layer directly.

This function should not be called while there are feature objects in existence that were obtained or created with the previous layer definition.

Not all drivers support this function. You can query a layer to check if it supports it with the OLCAlterFieldDefn capability. Certain drivers may only support this method while there are still no features in the layer. When it's supported, the existing features of the backing file/database should be updated accordingly. Certain drivers might also not support all update flags.

- iField: Index of the field whose definition must be altered.
- field\_def: New field definition
- nFlags: Combination of ALTER\_NAME\_FLAG, ALTER\_TYPE\_FLAG and ALTER\_WIDTH\_PRECISION\_FLAG to indicate which of the name and/or type and/or width and precision fields from the new field definition must be taken into account
- OGRERR\_NONE on success

Returns OGR 1.9.0.

**Clip(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr**

**CommitTransaction() -> OGRErr**

For data sources that support transactions, CommitTransaction commits a transaction.

If no transaction is active, or the commit fails, will return OGRERR\_FAILURE. Data sources that don't support transactions will always return OGRERR\_NONE.

Returns OGRERR\_NONE on success.

**CreateFeature(Feature feature) -> OGRErr**

Creates and writes a new feature within a layer.

The passed feature is written to the layer as a new feature, rather than overwriting an existing one. If the feature has a feature id other than OGRNullFID, then the native implementation may use that as the feature id of the new feature, but not necessarily. Upon successful return the passed feature will have been updated with the new feature id.

- feature: The handle of the feature to write to disk

Returns OGRERR\_NONE on success.



**CreateField(FieldDefn field\_def, int approx\_ok = 1) -> OGRErr**

Creates a new field on a layer.

You must use this to create new fields on a real layer. Internally the OGRFeatureDefn for the layer will be updated to reflect the new field. Applications should never modify the OGRFeatureDefn used by a layer directly.

This function shouldn't be called while there are feature objects in existence that were obtained or created with the previous layer definition.

Not all drivers support this function. You can query a layer to check if it supports it with the OLCCreateField capability. Certain drivers may only support this method while there are still no features in the layer. When it's supported, the existing features of the backing file/database should be updated accordingly.

- field\_def: Handle of the field definition to write to disk
- approx\_ok: If True, the field may be created in a slightly different form depending on the limitations of the format driver

Returns OGRERR\_NONE on success.

**CreateFields(fields)**

Creates a list of fields on the Layer.

**CreateGeomField(GeomFieldDefn field\_def, int approx\_ok = 1) -> OGRErr****DeleteFeature(long fid) -> OGRErr**

Deletes feature from layer.

- fid: The feature id to be deleted from the layer

Returns OGRERR\_NONE on success.

**DeleteField(int iField) -> OGRErr**

Creates a new field on a layer.

You must use this to delete existing fields on a real layer. Internally the OGRFeatureDefn for the layer will be updated to reflect the deleted field. Applications should never modify the OGRFeatureDefn used by a layer directly.

This function should not be called while there are feature objects in existence that were obtained or created with the previous layer definition.

Not all drivers support this function. You can query a layer to check if it supports it with the OLCDeleteField capability. Certain drivers may only support this method while there are still no features in the layer. When it's supported, the existing features of the backing file/database should be updated accordingly.

- iField: Index of the field to delete
- OGRERR\_NONE on success

Returns OGR 1.9.0.

**Dereference()**

For backward compatibility only.

**Erase(Layer method\_layer, Layer result\_layer, string options = None, GDALProgress-Func callback = 0, void callback\_data = None) -> OGRErr**

**FindFieldIndex(string pszFieldName, int bExactMatch) -> int**

**GetExtent(int force = 1, int can\_return\_null = 0, int geom\_field = 0)**

Fetches the extent of this layer.

Returns the extent (MBR) of the data in the layer. If bForce is False, and it would be expensive to establish the extent, then OGRERR\_FAILURE will be returned indicating that the extent isn't know. If bForce is True, then certain implementations will scan the entire layer once to compute the MBR of all the features in the layer.

Depending on the drivers, the returned extent may or may not take the spatial filter into account. It's safer to call OGR\_L\_GetExtent() without setting a spatial filter.

Layers without any geometry may return OGRERR\_FAILURE, indicating that no meaningful extents could be collected.

Note that certain implementations of this method may alter the read cursor of the layer.

- force: The structure in which the extent value will be returned
- can\_return\_null: Flag indicating whether the extent should be computed even if it's expensive

Returns OGRERR\_NONE on success, OGRERR\_FAILURE if extent not known.

**GetFIDColumn() -> char**

This method returns the name of the underlying database column being used as the FID column, or "" if not supported.

Returns fid column name.

**GetFeature(long fid) -> Feature**

Fetches a feature by its identifier.

This function will attempt to read the identified feature. The nFID value cannot be OGRNullFID. Success or failure of this operation is unaffected by the spatial or attribute filters.

If this function returns a non-None feature, it is guaranteed that its feature id (OGR\_F\_GetFID()) will be the same as nFID.

The returned feature should be free with OGR\_F\_Destroy().

- fid: The feature id of the feature to read

Returns a handle to a feature now owned by the caller or None on failure.

**GetFeatureCount(int force = 1) -> int**

Fetches the feature count in this layer.

Returns the number of features in the layer. For dynamic databases the count may not be exact. If bForce is False, and it would be expensive to establish the feature count, a value of -1 may be returned indicating that the count isn't known. If bForce is True, certain implementations will scan the entire layer once to count objects.

The returned count takes the spatial filter into account.

Note that certain implementations of this method may alter the read cursor of the layer.

- force: Flag indicating whether the count should be computed even if it's expensive.

Returns feature count, -1 if count not known.

**GetFeaturesRead() -> GIntBig****GetGeomType() -> OGRwkbGeometryType**

Returns the layer geometry type.

- The geometry type

Returns OGR 1.8.0.

**GetGeometryColumn() -> char**

This method returns the name of the underlying database column being used as the geometry column, or "" if not supported.

Returns geometry column name.

**GetLayerDefn() -> FeatureDefn**

Fetches the schema information for this layer.

The returned handle to the OGRFeatureDefn is owned by the OGRLayer, and shouldn't be modified or freed by the application. It encapsulates the attribute schema of the features of the layer.

Returns a handle to the feature definition.

**GetName() -> char**

Returns the layer name.

- The layer name (must not be freed)

Returns OGR 1.8.0.

**GetNextFeature() -> Feature**

Fetches the next available feature from this layer.

The returned feature becomes the responsibility of the caller to delete with `OGR_F_Destroy()`. It's critical that all features associated with an `OGR_Layer` (more specifically an `OGR_FeatureDefn`) be deleted before that layer/data source is deleted.

This function implements sequential access to the features of a layer. The `OGR_L_ResetReading()` function can be used to start at the beginning again.

Returns a handle to a feature or `None` if no more features are available.

**GetRefCount() -> int**

**GetSpatialFilter() -> Geometry**

This function returns the current spatial filter for this layer.

The returned pointer is to an internally owned object, and shouldn't be altered or deleted by the caller.

Returns a handle to the spatial filter geometry.

**GetSpatialRef() -> SpatialReference**

Fetches the spatial reference system for this layer.

The returned object is owned by the `OGR_Layer` and shouldn't be modified or freed by the application.

Returns spatial reference or `None` if there isn't one.

**GetStyleTable() -> StyleTable**

**Identity(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr**

**Intersection(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr**

**Reference()**

For backward compatibility only.

**ReorderField(int iOldFieldPos, int iNewFieldPos) -> OGRErr**

Reorders an existing field on a layer.

This function is a convenience wrapper of `OGR_L_ReorderFields()`, dedicated to move a single field.

You must use this to reorder existing fields on a real layer. Internally the `OGR_FeatureDefn` for the layer will be updated to reflect the reordering of the fields. Applications should never modify the `OGR_FeatureDefn` used by a layer directly.

This function shouldn't be called while there are feature objects in existence that were obtained or created with the previous layer definition.

The field definition that was at initial position `iOldFieldPos` will be moved at position `iNewFieldPos`, and elements between will be shuffled accordingly.

For example, suppose the fields were “0”, “1”, “2”, “3”, “4” initially. `ReorderField(1, 3)` will reorder them as “0”, “2”, “3”, “1”, “4”.

Not all drivers support this function. You can query a layer to check if it supports it with the `OLCReorderFields` capability. Certain drivers may only support this method while there are still no features in the layer. When it is supported, the existing features of the backing file/database should be updated accordingly.

- `iOldFieldPos`: Previous position of the field to move. Must be in the range `[0, GetFieldCount()-1]`.
- `iNewFieldPos`: New position of the field to move. Must be in the range `[0, GetFieldCount()-1]`.
- `OGRERR_NONE` on success.

Returns OGR 1.9.0.

### **ReorderFields(int nList) -> OGRErr**

Reorders all the fields of a layer.

You must use this to reorder existing fields on a real layer. Internally the `OGRFeatureDefn` for the layer will be updated to reflect the reordering of the fields. Applications should never modify the `OGRFeatureDefn` used by a layer directly.

This function should not be called while there are feature objects in existence that were obtained or created with the previous layer definition.

`panMap` is such that, for each field definition at position `i` after reordering, its position before reordering was `panMap[i]`.

For example, suppose the fields were “0”, “1”, “2”, “3”, “4” initially. `ReorderFields([0,2,3,1,4])` will reorder them as “0”, “2”, “3”, “1”, “4”.

Not all drivers support this function. You can query a layer to check if it supports it with the `OLCReorderFields` capability. Certain drivers may only support this method while there are still no features in the layer. When it’s supported, the existing features of the backing file/database should be updated accordingly.

- `nList`: An array of `GetLayerDefn()->GetFieldCount()` elements that’s a permutation of `[0, GetLayerDefn()->GetFieldCount()-1]`
- `OGRERR_NONE` on success

Returns OGR 1.9.0.

### **ResetReading()**

Resets feature reading to start on the first feature.

This affects `GetNextFeature()`.

### **RollbackTransaction() -> OGRErr**

For data sources that support transactions, `RollbackTransaction` will roll back a data source to its state before the start of the current transaction. If no transaction is

active, or the rollback fails, will return OGRERR\_FAILURE. Data sources that don't support transactions will always return OGRERR\_NONE.

Returns OGRERR\_NONE on success.

### **SetAttributeFilter(string filter\_string) -> OGRErr**

Sets a new attribute query.

This function sets the attribute query string to be used when fetching features via the OGR\_L\_GetNextFeature() function. Only features for which the query evaluates as true will be returned.

The query string should be in the format of a SQL WHERE clause; for example, "population > 1000000 and population < 5000000" where population is an attribute in the layer. The query format is a restricted form of SQL WHERE clause as defined "eq\_format=restricted\_where" about halfway through this document: <http://ogdi.sourceforge.net/prop/6.2.CapabilitiesMetadata.html>.

Note that installing a query string will generally result in resetting the current reading position (ala OGR\_L\_ResetReading()).

- filter\_string: Query in restricted SQL WHERE format, or None to clear the current query

Returns OGRERR\_NONE if successfully installed, or an error code if the query expression is in error, or another failure occurs.

### **SetFeature(Feature feature) -> OGRErr**

Rewrites an existing feature.

This function will write a feature to the layer, based on the feature id within the OGRFeature.

- feature: The feature to write

Returns OGRERR\_NONE if the operation works; otherwise, an appropriate error code.

### **SetIgnoredFields(string options) -> OGRErr**

Sets which fields can be omitted when retrieving features from the layer.

If the driver supports this functionality (testable using OLCIgnoreFields capability), it will not fetch the specified fields in subsequent calls to GetFeature() / GetNextFeature() and thus save processing time and/or bandwidth.

Besides field names of the layers, the following special fields can be passed: "OGR\_GEOMETRY" to ignore geometry and "OGR\_STYLE" to ignore layer style.

By default, no fields are ignored.

- options: An array of field names terminated by None item. If None is passed, the ignored list is cleared.

Returns OGRERR\_NONE if all field names have been resolved (even if the driver doesn't support this method).

**SetNextByIndex(long new\_index) -> OGRErr**

Moves read cursor to the nIndex'th feature in the current result set.

- new\_index: The index indicating how many steps into the result set to seek

Returns OGRERR\_NONE on success or an error code.

**SetSpatialFilter(int iGeomField, Geometry filter)**

Sets a new spatial filter.

This function sets the geometry to be used as a spatial filter when fetching features via the `OGR_L_GetNextFeature()` function. Only features that geometrically intersect the filter geometry will be returned.

This function makes an internal copy of the passed geometry. The passed geometry remains the responsibility of the caller and may be safely destroyed.

For the time being, the passed filter geometry should be in the same SRS as the layer (as returned by `OGR_L_GetSpatialRef()`). In the future this may be generalized.

- iGeomField: Handle to the geometry to use as a filtering region. None may be passed indicating that the current spatial filter should be cleared, but no new one instituted.

**SetSpatialFilterRect(int iGeomField, double minx, double miny, double maxx, double maxy)**

Sets a new rectangular spatial filter.

This method sets a rectangle to be used as a spatial filter when fetching features via the `OGR_L_GetNextFeature()` method. Only features that geometrically intersect the given rectangle will be returned.

The x/y values should be in the same coordinate system as the layer as a whole (as returned by `OGR_Layer::GetSpatialRef()`). Internally this method is normally implemented as creating a five-vertex closed rectangular polygon and passing it to `OGR_Layer::SetSpatialFilter()`. It exists as a convenience.

The only way to clear a spatial filter set with this method is to call `OGR_Layer::SetSpatialFilter(None)`.

- iGeomField: The minimum x coordinate for the rectangular region
- minx: The minimum y coordinate for the rectangular region
- miny: The maximum x coordinate for the rectangular region
- maxx: The maximum y coordinate for the rectangular region

**SetStyleTable(StyleTable table)****StartTransaction() -> OGRErr**

For data sources that support transactions, `StartTransaction` creates a transaction.

If starting the transaction fails, will return `OGRERR_FAILURE`. Data sources that don't support transactions will always return `OGRERR_NONE`.

Returns OGRERR\_NONE on success.

**SymDifference(Layer method\_layer, Layer result\_layer, string options = None, GDAL-ProgressFunc callback = 0, void callback\_data = None) -> OGRErr**

**SyncToDisk() -> OGRErr**

Flushes pending changes to disk.

This call is intended to force the layer to flush any pending writes to disk, and leave the disk file in a consistent state. It wouldn't normally have any effect on read-only data sources.

Certain layers don't implement this method, and will still return OGRERR\_NONE. The default implementation returns OGRERR\_NONE. An error is only returned if an error occurs while attempting to flush to disk.

Returns OGRERR\_NONE if no error occurs (even if nothing is done) or an error code.

**TestCapability(string cap) -> bool**

Tests if this layer supported the named capability.

The capability codes that can be tested are represented as strings, but defined constants exist to ensure correct spelling. Specific layer types may implement class specific capabilities, but this can't generally be discovered by the caller.

OLCRandomRead / "RandomRead": True if the GetFeature() method is implemented in an optimized way for this layer, as opposed to the default implementation using ResetReading() and GetNextFeature() to find the requested feature id.

OLCSequentialWrite / "SequentialWrite": True if the CreateFeature() method works for this layer. Note this means that this particular layer is writable. The same OGRLayer class may return False for other layer instances that are effectively read-only.

OLCRandomWrite / "RandomWrite": True if the SetFeature() method is operational on this layer. Note this means that this particular layer is writable. The same OGRLayer class may return False for other layer instances that are effectively read-only.

OLCFastSpatialFilter / "FastSpatialFilter": True if this layer implements spatial filtering efficiently. Layers that effectively read all features, and test them with the OGRFeature intersection methods should return False. This can be used as a clue by the application whether it should build and maintain its own spatial index for features in this layer.

OLCFastFeatureCount / "FastFeatureCount": True if this layer can return a feature count (via OGR\_L\_GetFeatureCount()) efficiently—without counting the features. In certain cases this will return True until a spatial filter is installed after which it will return False.

OLCFastGetExtent / "FastGetExtent": True if this layer can return its data extent (via OGR\_L\_GetExtent()) efficiently—without scanning all the features. In some cases this will return True until a spatial filter is installed after which it will return False.



OLCFastSetNextByIndex / “FastSetNextByIndex”: True if this layer can perform the SetNextByIndex() call efficiently; otherwise, False.

OLCCreateField / “CreateField”: True if this layer can create new fields on the current layer using CreateField(); otherwise, False.

OLCDeleteField / “DeleteField”: True if this layer can delete existing fields on the current layer using DeleteField(); otherwise, False.

OLCDeleteFeature / “DeleteFeature”: True if the DeleteFeature() method is supported on this layer; otherwise, False.

OLCStringsAsUTF8 / “StringsAsUTF8”: True if values of OFTString fields are assured to be in UTF-8 format. If False, the encoding of fields is uncertain, though it might still be UTF-8.

- cap: The name of the capability to test.

Returns True if the layer has the requested capability, or False otherwise. OGRLayers will return False for any unrecognized capabilities.

Union(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr

Update(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr

Update(Layer method\_layer, Layer result\_layer, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> OGRErr

next()

## PROPERTIES

schema

## C.10 *StyleTable class*

AddStyle(string pszName, string pszStyleString) -> int

Find(string pszName) -> char

GetLastStyleName() -> char

GetNextStyle() -> char

LoadStyleTable(string utf8\_path) -> int

ResetStyleStringReading()

SaveStyleTable(string utf8\_path) -> int

SaveStyleTable(string utf8\_path) -> int

## **C.11 Module functions**

ApproximateArcAngles(double dfCenterX, double dfCenterY, double dfZ, double dfPrimaryRadius, double dfSecondaryAxis, double dfRotation, double dfStartAngle, double dfEndAngle, double dfMaxAngleStepSizeDegrees) -> Geometry

BuildPolygonFromEdges(Geometry hLineCollection, int bBestEffort = 0, int bAutoClose = 0, double dfTolerance = 0) -> Geometry

CreateGeometryFromGML(char input\_string) -> Geometry

CreateGeometryFromJson(char input\_string) -> Geometry

CreateGeometryFromWkb(int len, SpatialReference reference = None) -> Geometry

CreateGeometryFromWkt(char val, SpatialReference reference = None) -> Geometry

DontUseExceptions()

ForceToLineString(Geometry geom\_in) -> Geometry

ForceToMultiLineString(Geometry geom\_in) -> Geometry

ForceToMultiPoint(Geometry geom\_in) -> Geometry

ForceToMultiPolygon(Geometry geom\_in) -> Geometry

ForceToPolygon(Geometry geom\_in) -> Geometry

GeneralCmdLineProcessor(char papszArgv, int nOptions = 0) -> char

GeometryTypeToName(OGRwkbGeometryType eType) -> char

GetDriver(int driver\_number) -> Driver

GetDriverByName(char name) -> Driver

GetDriverCount() -> int

GetFieldName(OGRFieldType type) -> char

**GetOpenDS(int ds\_number) -> DataSource**

**GetOpenDSCount() -> int**

**GetUseExceptions() -> int**

**Open(char utf8\_path, int update = 0) -> DataSource**

**OpenShared(char utf8\_path, int update = 0) -> DataSource**

**RegisterAll()**

**SetGenerate\_DB2\_V72\_BYTE\_ORDER(int bGenerate\_DB2\_V72\_BYTE\_ORDER) -> OGRErr**

**TermProgress\_nocb(double dfProgress, string pszMessage = None, void pData = None) -> int**

**UseExceptions()**

## **C.12 Module data**

ALTER_ALL_FLAG	OFTWideStringList
ALTER_NAME_FLAG	OJLeft
ALTER_TYPE_FLAG	OJRight
ALTER_WIDTH_PRECISION_FLAG	OJUndefined
NullFID	OLCAlterFieldDefn
ODrCCreateDataSource	OLCCreateField
ODrCDeleteDataSource	OLCCreateGeomField
ODsCCreateGeomFieldAfterCreateLayer	OLCDeleteFeature
ODsCCreateLayer	OLCDeleteField
ODsCDeleteLayer	OLCFastFeatureCount
OFTBinary	OLCFastGetExtent
OFTDate	OLCFastSetNextByIndex
OFTDateTime	OLCFastSpatialFilter
OFTInteger	OLCIgnoreFields
OFTIntegerList	OLCRandomRead
OFTReal	OLCRandomWrite
OFTRealList	OLCReorderFields
OFTString	OLCSequentialWrite
OFTStringList	OLCStringsAsUTF8
OFTTime	OLCTransactions
OFTWideString	TermProgress

wkb25Bit	wkbMultiPolygon
wkb25DBit	wkbMultiPolygon25D
wkbGeometryCollection	wkbNDR
wkbGeometryCollection25D	wkbNone
wkbLineString	wkbPoint
wkbLineString25D	wkbPoint25D
wkbLinearRing	wkbPolygon
wkbMultiLineString	wkbPolygon25D
wkbMultiLineString25D	wkbUnknown
wkbMultiPoint	wkbXDR
wkbMultiPoint25D	



# *appendix D*

## *OSR*

---

### **D.1 Summary**

#### **COORDINATETRANSFORMATION CLASS**

TransformPoint()  
TransformPoints()  
TransformPoints()

#### **SPATIALREFERENCE CLASS**

AutoIdentifyEPSG()  
Clone()  
CloneGeogCS()  
CopyGeogCSFrom()  
EPSGTreatsAsLatLong()  
EPSGTreatsAsNorthingEasting()  
ExportToMICoordSys()  
ExportToPCI()  
ExportToPrettyWkt()  
ExportToProj4()  
ExportToUSGS()  
ExportToWkt()  
ExportToXML()  
Fixup()  
FixupOrdering()  
GetsAngularUnits()  
GetsAttrValue()  
GetsAuthorityCode()  
GetsAuthorityName()

GetsInvFlattening()  
GetsLinearUnits()  
GetsLinearUnitsName()  
GetsNormProjParm()  
GetsProjParm()  
GetsSemiMajor()  
GetsSemiMinor()  
GetsTOWGS84()  
GetsUTMZone()  
ImportFromEPSG()  
ImportFromEPSGA()  
ImportFromERM()  
ImportFromESRI()  
ImportFromMICoordSys()  
ImportFromOzi()  
ImportFromPCI()  
ImportFromProj4()  
ImportFromUSGS()  
ImportFromUrl()  
ImportFromWkt()  
ImportFromXML()  
IsCompound()  
IsGeocentric()  
IsGeographic()  
IsLocal()

IsProjected()  
 IsSame()  
 IsSameGeogCS()  
 IsSameVertCS()  
 IsVertical()  
 MorphFromESRI()  
 MorphToESRI()  
 SetACEA()  
 SetAE()  
 SetAngularUnits()  
 SetAttrValue()  
 SetAuthority()  
 SetBonne()  
 SetCEA()  
 SetCS()  
 SetCompoundCS()  
 SetEC()  
 SetEckertIV()  
 SetEckertVI()  
 SetEquirectangular()  
 SetEquirectangular2()  
 SetFromUserInput()  
 SetGEOS()  
 SetGH()  
 SetGS()  
 SetGaussSchreiberTMercator()  
 SetGeocCS()  
 SetGeogCS()  
 SetGnomonic()  
 SetHOM()  
 SetHOM2PNO()  
 SetIGH()  
 SetKrovak()  
 SetLAEA()  
 SetLCC()  
 SetLCC1SP()  
 SetLCCB()  
 SetLinearUnits()  
 SetLinearUnitsAndUpdateParameters()

SetLocalCS()  
 SetMC()  
 SetMercator()  
 SetMollweide()  
 SetNZMG()  
 SetNormProjParm()  
 SetOS()  
 SetOrthographic()  
 SetPS()  
 SetPolyconic()  
 SetProjCS()  
 SetProjParm()  
 SetProjection()  
 SetRobinson()  
 SetSOC()  
 SetSinusoidal()  
 SetStatePlane()  
 SetStereographic()  
 SetTM()  
 SetTMG()  
 SetTMSO()  
 SetTMVariant()  
 SetTOWGS84()  
 SetTargetLinearUnits()  
 SetUTM()  
 SetVDG()  
 SetVertCS()  
 SetWellKnownGeogCS()  
 StripCTParms()  
 Validate()  
 Validate()

#### **MODULE FUNCTIONS**

CreateCoordinateTransformation()  
 DontUseExceptions()  
 GetsProjectionMethods()  
 GetsUseExceptions()  
 GetUserInputAsWKT()  
 GetsWellKnownGeogCSAsWKT()  
 UseExceptions()

## D.2 **CoordinateTransformation class**

**TransformPoint(double x, double y, double z = 0.0)**

**TransformPoints(int nCount)**

**TransformPoints(int nCount)**

## D.3 **SpatialReference class**

**AutoIdentifyEPSG() -> OGRErr**

Sets EPSG authority information if possible.

This method inspects a WKT definition and adds EPSG authority nodes where an aspect of the coordinate system can be easily and safely corresponded with an EPSG identifier. In practice, this method will evolve over time. In theory, it can add authority nodes for any object (that is, spheroid, datum, GEOGCS, units, and PROJCS) that could have an authority node. Mostly this is useful for inserting appropriate PROJCS codes for common formulations (such as UTM and WGS84).

If it's successful the OGRSpatialReference is updated in place, and the method returns OGRERR\_NONE. If the method fails to identify the general coordinate system, OGRERR\_UNSUPPORTED\_SRS is returned but no error message is posted via CPL-Error().

Returns OGRERR\_NONE or OGRERR\_UNSUPPORTED\_SRS.

**Clone() -> SpatialReference**

Makes a duplicate of this OGRSpatialReference.

Returns a new SRS, which becomes the responsibility of the caller.

**CloneGeogCS() -> SpatialReference**

Makes a duplicate of the GEOGCS node of this OGRSpatialReference object.

Returns a new SRS, which becomes the responsibility of the caller.

**CopyGeogCSFrom(SpatialReference rhs) -> OGRErr**

Copies GEOGCS from another OGRSpatialReference.

The GEOGCS information is copied into this OGRSpatialReference from another. If this object has a PROJCS root already, the GEOGCS is installed within it; otherwise, it's installed as the root.

- poSrcSRS: The spatial reference to copy the GEOGCS information from

Returns OGRERR\_NONE on success or an error code.

**EPSGTreatsAsLatLong() -> int**

This method returns TRUE if EPSG feels this geographic coordinate system should be treated as having lat/long coordinate ordering.



Currently this returns TRUE for all geographic coordinate systems with an EPSG code set, and AXIS values set defining it as lat, long. Note that coordinate systems with an EPSG code and no axis settings will be assumed to not be lat/long.

FALSE will be returned for all coordinate systems that are not geographic, or that don't have an EPSG code set.

Returns TRUE or FALSE.

#### **EPSTreatsAsNorthingEasting() -> int**

This method returns TRUE if EPSG feels this projected coordinate system should be treated as having northing/easting coordinate ordering.

Currently this returns TRUE for all projected coordinate systems with an EPSG code set, and AXIS values set defining it as northing, easting.

FALSE will be returned for all coordinate systems that aren't projected, or that don't have an EPSG code set.

Returns TRUE or FALSE.

Since OGR 1.10.0.

#### **ExportToMICoordSys() -> OGRErr**

#### **ExportToPCI() -> OGRErr**

#### **ExportToPrettyWkt(int simplify = 0) -> OGRErr**

#### **ExportToProj4() -> OGRErr**

#### **ExportToUSGS() -> OGRErr**

#### **ExportToWkt() -> OGRErr**

#### **ExportToXML(string dialect = "") -> OGRErr**

#### **Fixup() -> OGRErr**

Fixes up as needed.

Certain mechanisms to create WKT using OGRSpatialReference and some imported WKT aren't valid according to the OGC CT specification. This method attempts to fill in any missing defaults that are required, and fix up ordering problems (using OSR-FixupOrdering()) so that the resulting WKT is valid.

This method should be expected to evolve over time as problems are discovered. The following are among the fix-up actions this method will take:

- Fix up the ordering of nodes to match the BNF WKT ordering, using the FixupOrdering() method.
- Add missing linear or angular units nodes.

Returns OGRERR\_NONE on success or an error code if something goes wrong.

### **FixupOrdering() -> OGRErr**

Corrects parameter ordering to match CT Specification.

Certain mechanisms to create WKT using OGRSpatialReference and some imported WKT fail to maintain the order of parameters required, according to the BNF definitions in the OpenGIS SF-SQL and CT Specifications. This method attempts to massage things back into the required order.

Returns OGRERR\_NONE on success or an error code if something goes wrong.

### **GetsAngularUnits() -> double**

Fetches angular geographic coordinate system units.

If no units are available, a value of “degree” and SRS\_UA\_DEGREE\_CONV will be assumed. This method only checks directly under the GEOGCS node for units.

This method does the same thing as the C function OSRGetsAngularUnits().

- ppszName: A pointer to be updated with the pointer to the unit’s name. The returned value remains internal to the OGRSpatialReference and shouldn’t be freed or modified. It may be invalidated on the next OGRSpatialReference call.

Returns the value to multiply by angular distances to transform them to radians.

### **GetsAttrValue(string name, int child = 0) -> char**

Fetches indicated attribute of named node.

This method uses GetsAttrNode() to find the named node, and then extracts the value of the indicated child. Thus, a call to GetsAttrValue(“UNIT”,1) would return the second child of the UNIT node, which is normally the length of the linear unit in meters.

This method does the same thing as the C function OSRGetsAttrValue().

- pszNodeName: The tree node to look for (case insensitive)
- iAttr: The child of the node to fetch (zero-based)

Returns the requested value, or NULL if it fails for any reason.

### **GetsAuthorityCode(string target\_key) -> char**

Gets the authority code for a node.

This method is used to query an AUTHORITY[] node from within the WKT tree and fetch the code value.

While in theory values may be non-numeric, for the EPSG authority all code values should be integral.

- pszTargetKey: The partial or complete path to the node to get an authority from—that is, “PROJCS”, “GEOGCS”, “GEOGCS|UNIT”, or NULL to search for an authority node on the root element.

Returns value code from authority node, or NULL on failure. The value returned is internal and should not be freed or modified.

### **GetsAuthorityName(string target\_key) -> char**

Gets the authority name for a node.

This method is used to query an AUTHORITY[] node from within the WKT tree, and fetch the authority name value.

The most common authority is EPSG.

- pszTargetKey: The partial or complete path to the node to get an authority from. That is, "PROJCS", "GEOGCS", "GEOGCS|UNIT", or NULL to search for an authority node on the root element.

Returns value code from authority node, or NULL on failure. The value returned is internal and should not be freed or modified.

### **GetsInvFlattening() -> double**

Gets spheroid inverse flattening.

This method does the same thing as the C function OSRGetsInvFlattening().

- pnErr: If non-NULL set to OGRERR\_FAILURE if no inverse flattening can be found.

Returns inverse flattening, or SRS\_WGS84\_INVFLATTENING if it can't be found.

### **GetsLinearUnits() -> double**

Fetches linear projection units.

If no units are available, a value of "Meters" and 1.0 will be assumed. This method only checks directly under the PROJCS, GEOGCS, or LOCAL\_CS node for units.

This method does the same thing as the C function OSRGetsLinearUnits()/

- ppszName: A pointer to be updated with the pointer to the unit's name. The returned value remains internal to the OGRSpatialReference and shouldn't be freed or modified. It may be invalidated on the next OGRSpatialReference call.

Returns the value to multiply by linear distances to transform them to meters.

### **GetsLinearUnitsName() -> char**

### **GetsNormProjParm(string name, double default\_val = 0.0) -> double**

Fetches a normalized projection parameter value.

- pszName: The name of the parameter to fetch, from the set of SRS\_PP codes in ogr\_srs\_api.h.
- dfDefaultValue: The value to return if this parameter doesn't exist.
- pnErr: Place to put error code on failure. Ignored if NULL.

Returns value of parameter.

**GetsProjParm(string name, double default\_val = 0.0) -> double**

Fetches a projection parameter value.

NOTE: This code should be modified to translate non-degree angles into degrees based on the GEOGCS unit. This hasn't yet been done.

- pszName: The name of the parameter to fetch, from the set of SRS\_PP codes in ogr\_srs\_api.h.
- dfDefaultValue: The value to return if this parameter doesn't exist.
- pnErr: Place to put error code on failure. Ignored if NULL.

Returns value of parameter.

**GetsSemiMajor() -> double**

Gets spheroid semi-major axis.

This method does the same thing as the C function OSRGetsSemiMajor().

- pnErr: When non-NULL set to OGRERR\_FAILURE if semi major axis can be found

Returns semi-major axis, or SRS\_WGS84\_SEMIMAJOR if it can't be found.

**GetsSemiMinor() -> double**

Gets spheroid semi-minor axis.

This method does the same thing as the C function OSRGetsSemiMinor().

- pnErr: When non-NULL set to OGRERR\_FAILURE if semi minor axis can be found

Returns semi-minor axis, or WGS84 semi-minor if it can't be found.

**GetsTOWGS84() -> OGRErr**

Fetches TOWGS84 parameters, if available.

- padfCoeff: Array into which up to seven coefficients are placed
- nCoeffCount: Size of padfCoeff—defaults to 7

Returns OGRERR\_NONE on success, or OGRERR\_FAILURE if there's no TOWGS84 node available.

**GetsUTMZone() -> int**

Gets utm zone information.

This is the same as the C function OSRGetsUTMZone().

In SWIG bindings (Python, Java, etc.) the GetsUTMZone() method returns a zone that's negative in the southern hemisphere instead of having the pbNorth flag used in the C and C++ interface.

- pbNorth: Pointer to set to TRUE if northern hemisphere, or FALSE if southern.

Returns UTM zone number or zero if this isn't a UTM definition.

**ImportFromEPSG(int arg) -> OGRErr**

**ImportFromEPSGA(int arg) -> OGRErr**

**ImportFromERM(string proj, string datum, string units) -> OGRErr**

**ImportFromESRI(string ppszInput) -> OGRErr**

**ImportFromMICoordSys(string pszCoordSys) -> OGRErr**

**ImportFromOzi(string datum, string proj, string projParms) -> OGRErr**

**ImportFromPCI(string proj, string units = "METRE", double argin = 0) -> OGRErr**

**ImportFromProj4(string ppszInput) -> OGRErr**

**ImportFromUSGS(long proj\_code, long zone = 0, double argin = 0, long datum\_code = 0) -> OGRErr**

**ImportFromUrl(string url) -> OGRErr**

**ImportFromWkt(string ppszInput) -> OGRErr**

**ImportFromXML(string xmlString) -> OGRErr**

**IsCompound() -> int**

Checks if coordinate system is compound.

Returns TRUE if this is rooted with a COMPD\_CS node.

**IsGeocentric() -> int**

Checks if geocentric coordinate system.

Returns TRUE if this contains a GEOCCS node indicating it's a geocentric coordinate system.

Since OGR 1.9.0.

**IsGeographic() -> int**

Checks if geographic coordinate system.

Returns TRUE if this spatial reference is geographic—that is, if the root is a GEOGCS node.

**IsLocal() -> int**

Checks if local coordinate system.

Returns TRUE if this spatial reference is local—that is, if the root is a LOCAL\_CS node.

#### **IsProjected() -> int**

Checks if projected coordinate system.

Returns TRUE if this contains a PROJCS node indicating it's a projected coordinate system.

#### **IsSame(SpatialReference rhs) -> int**

Do these two spatial references describe the same system?

- poOtherSRS: The SRS being compared to

Returns TRUE if equivalent or FALSE otherwise.

#### **IsSameGeogCS(SpatialReference rhs) -> int**

Do the GeogCSs match?

- poOther: The SRS being compared against

Returns TRUE if they're the same or FALSE otherwise.

#### **IsSameVertCS(SpatialReference rhs) -> int**

Do the VertCSs match?

- poOther: The SRS being compared against

Returns TRUE if they are the same or FALSE otherwise.

#### **IsVertical() -> int**

Checks if vertical coordinate system.

Returns TRUE if this contains a VERT\_CS node indicating it's a vertical coordinate system.

Since OGR 1.8.0.

#### **MorphFromESRI() -> OGRErr**

#### **MorphToESRI() -> OGRErr**

**SetACEA(double stdp1, double stdp2, double clat, double clong, double fe, double fn) -> OGRErr**

**SetAE(double clat, double clong, double fe, double fn) -> OGRErr**

**SetAngularUnits(string name, double to\_radians) -> OGRErr**

Sets the angular units for the geographic coordinate system.

This method creates a UNIT subnode with the specified values as a child of the GEOGCS node.

This method does the same as the C function `OSRSetAngularUnits()`.

- `pszUnitsName`: The unit's name to be used. Several preferred unit names can be found in `ogr_srs_api.h`, such as `SRS_UA_DEGREE`.
- `dfInRadians`: The value to multiply by an angle in the indicated units to transform to radians. Several standard conversion factors can be found in `ogr_srs_api.h`.

Returns `OGRERR_NONE` on success.

**SetAttrValue(string name, string value) -> OGRErr**

**SetAuthority(string pszTargetKey, string pszAuthority, int nCode) -> OGRErr**

Sets the authority for a node.

- `pszTargetKey`: The partial or complete path to the node to set an authority on. That is, "PROJCS", "GEOGCS", or "GEOGCS|UNIT".
- `pszAuthority`: Authority name, such as "EPSG".
- `nCode`: Code for value with this authority.

Returns `OGRERR_NONE` on success.

**SetBonne(double stdp, double cm, double fe, double fn) -> OGRErr**

**SetCEA(double stdp1, double cm, double fe, double fn) -> OGRErr**

**SetCS(double clat, double clong, double fe, double fn) -> OGRErr**

**SetCompoundCS(string name, SpatialReference horizcs, SpatialReference vertcs) -> OGRErr**

Sets up a compound coordinate system.

This method replaces the current SRS with a `COMPD_CS` coordinate system consisting of the passed-in horizontal and vertical coordinate systems.

- `pszName`: The name of the compound coordinate system
- `poHorizSRS`: The horizontal SRS (`PROJCS` or `GEOGCS`)
- `poVertSRS`: The vertical SRS (`VERT_CS`)

Returns `OGRERR_NONE` on success.

**SetEC(double stdp1, double stdp2, double clat, double clong, double fe, double fn) -> OGRErr**

**SetEckertIV(double cm, double fe, double fn) -> OGRErr**

**SetEckertVI(double cm, double fe, double fn) -> OGRErr**

**SetEquirectangular(double clat, double clong, double fe, double fn) -> OGRErr**

**SetEquirectangular2(double clat, double clong, double pseudostdparallellat, double fe, double fn) -> OGRErr**

**SetFromUserInput(string name) -> OGRErr**

Sets spatial reference from various text formats.

This method will examine the provided input and try to deduce the format, and then use it to initialize the spatial reference system. It may take the following forms:

- Well-known text definition - passed on to `importFromWkt()`
- "EPSG:n"—number passed on to `importFromEPSG()`
- "EPSGA:n"—number passed on to `importFromEPSGA()`
- "AUTO:proj\_id,unit\_id,lon0,lat0"—WMS auto projections
- "urn:ogc:def:crs:EPSG::n"—ogc urns
- PROJ.4 definitions—passed on to `importFromProj4()`
- Filename—file read for WKT, XML or PROJ.4 definition
- Well-known name accepted by `SetWellKnownGeogCS()`, such as NAD27, NAD83, WGS84 or WGS72
- WKT (directly or in a file) in ESRI format should be prefixed with `ESRI::` to trigger an automatic `morphFromESRI()`
- "IGNF:xxx"—"+init=IGNF:xxx" passed on to `importFromProj4()`

It's expected that this method will be extended in the future to support XML and perhaps a simplified "mini-language" for indicating common UTM and state plane definitions.

This method is intended to be flexible, but by its nature it's imprecise because it must guess information about the format intended. When possible, applications should call the specific method appropriate if the input is known to be in a particular format.

This method does the same thing as the `OSRSetFromUserInput()` function.

- `pszDefinition`: Text definition to try to deduce SRS from

Returns `OGRErr_NONE` on success, or an error code if the name isn't recognized, the definition is corrupt, or an EPSG value can't be successfully looked up.

**SetGEOS(double cm, double satelliteheight, double fe, double fn) -> OGRErr**

**SetGH(double cm, double fe, double fn) -> OGRErr**

**SetGS(double cm, double fe, double fn) -> OGRErr**



**SetGaussSchreiberTMercator(double clat, double clong, double sc, double fe, double fn) -> OGRErr**

**SetGeocCS(string name = "unnamed") -> OGRErr**

Sets the user-visible GEOCCS name.

This method will ensure a GEOCCS node is created as the root, and set the provided name on it. If used on a GEOGCS coordinate system, the DATUM and PRIMEM nodes from the GEOGCS will be transferred over to the GEOCCS.

- pszName: The user-visible name to assign. Not used as a key.

Returns OGRERR\_NONE on success.

Since OGR 1.9.0.

**SetGeogCS(string pszGeogName, string pszDatumName, string pszEllipsoidName, double dfSemiMajor, double dfInvFlattening, string pszPMName = "Greenwich", double dfPMOffset = 0.0, string pszUnits = "degree", double dfConvertToRadians = 0.0174532925199433) -> OGRErr**

Sets geographic coordinate system.

This method is used to set the datum, ellipsoid, prime meridian, and angular units for a geographic coordinate system. It can be used on its own to establish a geographic spatial reference, or applied to a projected coordinate system to establish the underlying geographic coordinate system.

This method does the same as the C function OSRSetGeogCS().

- pszGeogName: User visible name for the geographic coordinate system (not to serve as a key).
- pszDatumName: Key name for this datum. The OpenGIS specification lists known values, and otherwise EPSG datum names with a standard transformation are considered legal keys.
- pszSpheroidName: User visible spheroid name (not to serve as a key).
- dfSemiMajor: The semi-major axis of the spheroid.
- dfInvFlattening: The inverse flattening for the spheroid. This can be computed from the semi-minor axis as  $1/f = 1.0 / (1.0 - \text{semiminor}/\text{semimajor})$ .
- pszPMName: The name of the prime meridian (not to serve as a key). If this is NULL, a default value of "Greenwich" will be used.
- dfPMOffset: The longitude of Greenwich relative to this prime meridian.
- pszAngularUnits: The angular unit's name (see ogr\_srs\_api.h for some standard names). If NULL, a value of "degrees" will be assumed.
- dfConvertToRadians: Value to multiply angular units by to transform them to radians. A value of SRS\_UL\_DEGREE\_CONV will be used if pszAngularUnits is NULL.

Returns OGRERR\_NONE on success.

**SetGnomonic(double clat, double clong, double fe, double fn) -> OGRErr**

**SetHOM(double clat, double clong, double azimuth, double recttoskew, double scale, double fe, double fn) -> OGRErr**

Sets a Hotine Oblique Mercator projection using azimuth angle.

This projection corresponds to EPSG projection method 9812, also sometimes known as *Hotine Oblique Mercator (variant A)*.

This method does the same thing as the C function OSRSetHOM().

- dfCenterLat: Latitude of the projection origin
- dfCenterLong: Longitude of the projection origin
- dfAzimuth: Azimuth, measured clockwise from North, of the projection center-line
- dfRectToSkew: Angle from rectified to skew grid
- dfScale: Scale factor applies to the projection origin
- dfFalseEasting: False easting
- dfFalseNorthing: False northing

Returns OGRERR\_NONE on success.

**SetHOM2PNO(double clat, double dfLat1, double dfLong1, double dfLat2, double dfLong2, double scale, double fe, double fn) -> OGRErr**

Sets a Hotine Oblique Mercator projection using two points on projection center-line.

This method does the same thing as the C function OSRSetHOM2PNO().

- dfCenterLat: Latitude of the projection origin
- dfLat1: Latitude of the first point on center line
- dfLong1: Longitude of the first point on center line
- dfLat2: Latitude of the second point on center line
- dfLong2: Longitude of the second point on center line
- dfScale: Scale factor applies to the projection origin
- dfFalseEasting: False easting
- dfFalseNorthing: False northing

Returns OGRERR\_NONE on success.

**SetIGH() -> OGRErr**

**SetKrovak(double clat, double clong, double azimuth, double pseudostdparallelat, double scale, double fe, double fn) -> OGRErr**

**SetLAEA(double clat, double clong, double fe, double fn) -> OGRErr**

**SetLCC(double stdp1, double stdp2, double clat, double clong, double fe, double fn)**  
-> OGRErr

**SetLCC1SP(double clat, double clong, double scale, double fe, double fn)** -> OGRErr

**SetLCCB(double stdp1, double stdp2, double clat, double clong, double fe, double fn)** -> OGRErr

**SetLinearUnits(string name, double to\_meters)** -> OGRErr

Sets the linear units for the projection.

This method creates a UNIT subnode with the specified values as a child of the PROJCS, GEOCCS, or LOCAL\_CS node.

This method does the same as the C function OSRSetLinearUnits().

- pszUnitsName: The unit's name to be used. Several preferred unit names can be found in ogr\_srs\_api.h, such as SRS\_UL\_METER, SRS\_UL\_FOOT, and SRS\_UL\_US\_FOOT.
- dfInMeters: The value to multiply by a length in the indicated units to transform to meters. Several standard conversion factors can be found in ogr\_srs\_api.h.

Returns OGRErr\_NONE on success.

**SetLinearUnitsAndUpdateParameters(string name, double to\_meters)** -> OGRErr

Sets the linear units for the projection.

This method creates a UNIT subnode with the specified values as a child of the PROJCS or LOCAL\_CS node. It works the same as the SetLinearUnits() method, but it also updates all existing linear projection parameter values from the old units to the new units.

- pszName: The unit's name to be used. Several preferred unit names can be found in ogr\_srs\_api.h, such as SRS\_UL\_METER, SRS\_UL\_FOOT, and SRS\_UL\_US\_FOOT.
- dfInMeters: The value to multiply by a length in the indicated units to transform to meters. Several standard conversion factors can be found in ogr\_srs\_api.h.

Returns OGRErr\_NONE on success.

**SetLocalCS(string pszName)** -> OGRErr

Sets the user-visible LOCAL\_CS name.

This method will ensure a LOCAL\_CS node is created as the root, and set the provided name on it. It must be used before SetLinearUnits().

- pszName: The user-visible name to assign. Not used as a key.

Returns OGRErr\_NONE on success.

**SetMC(double clat, double clong, double fe, double fn) -> OGRErr**

**SetMercator(double clat, double clong, double scale, double fe, double fn) -> OGRErr**

**SetMollweide(double cm, double fe, double fn) -> OGRErr**

**SetNZMG(double clat, double clong, double fe, double fn) -> OGRErr**

**SetNormProjParm(string name, double val) -> OGRErr**

Sets a projection parameter with a normalized value.

- pszName: The parameter name, which should be selected from the macros in ogr\_srs\_api.h, such as SRS\_PP\_CENTRAL\_MERIDIAN.
- dfValue: Value to assign.

Returns OGRERR\_NONE on success.

**SetOS(double dfOriginLat, double dfCMeridian, double scale, double fe, double fn) -> OGRErr**

**SetOrthographic(double clat, double clong, double fe, double fn) -> OGRErr**

**SetPS(double clat, double clong, double scale, double fe, double fn) -> OGRErr**

**SetPolyconic(double clat, double clong, double fe, double fn) -> OGRErr**

**SetProjCS(string name = "unnamed") -> OGRErr**

Sets the user-visible PROJCS name.

This method will ensure a PROJCS node is created as the root, and set the provided name on it. If used on a GEOGCS coordinate system, the GEOGCS node will be demoted to be a child of the new PROJCS root.

- pszName: The user-visible name to assign. Not used as a key.

Returns OGRERR\_NONE on success.

**SetProjParm(string name, double val) -> OGRErr**

Sets a projection parameter value.

Adds a new PARAMETER under the PROJCS with the indicated name and value.

Please check [http://www.remotesensing.org/geotiff/proj\\_list](http://www.remotesensing.org/geotiff/proj_list) pages for legal parameter names for specific projections.

- pszParmName: The parameter name, which should be selected from the macros in ogr\_srs\_api.h, such as SRS\_PP\_CENTRAL\_MERIDIAN.
- dfValue: Value to assign.

Returns OGRERR\_NONE on success.

### **SetProjection(string arg) -> OGRErr**

Sets a projection name.

- pszProjection: The projection name, which should be selected from the macros in ogr\_srs\_api.h, such as SRS\_PT\_TRANSVERSE\_MERCATOR.

Returns OGRERR\_NONE on success.

### **SetRobinson(double clong, double fe, double fn) -> OGRErr**

### **SetSOC(double latitudeoforigin, double cm, double fe, double fn) -> OGRErr**

### **SetSinusoidal(double clong, double fe, double fn) -> OGRErr**

### **SetStatePlane(int zone, int is\_nad83 = 1, string unitsname = "", double units = 0.0) -> OGRErr**

State plane.

Sets state plane projection definition.

This will attempt to generate a complete definition of a state plane zone based on generating the entire SRS from the EPSG tables. If the EPSG tables are unavailable, it will produce a stubbed LOCAL\_CS definition and return OGRERR\_FAILURE.

- nZone: State plane zone number, in the USGS numbering scheme (as distinct from the Arc/Info and Erdas numbering scheme).
- bNAD83: TRUE if the NAD83 zone definition should be used, or FALSE if the NAD27 zone definition should be used.
- pszOverrideUnitName: Linear unit name to apply overriding the legal definition for this zone.
- dfOverrideUnit: Linear unit conversion factor to apply overriding the legal definition for this zone.

Returns OGRERR\_NONE on success or OGRERR\_FAILURE on failure, mostly likely due to the EPSG tables not being accessible.

### **SetStereographic(double clat, double clong, double scale, double fe, double fn) -> OGRErr**

### **SetTM(double clat, double clong, double scale, double fe, double fn) -> OGRErr**

### **SetTMG(double clat, double clong, double fe, double fn) -> OGRErr**

### **SetTMSO(double clat, double clong, double scale, double fe, double fn) -> OGRErr**

**SetTMVariant(string pszVariantName, double clat, double clong, double scale, double fe, double fn) -> OGRErr**

Transverse Mercator variants.

**SetTOWGS84(double p1, double p2, double p3, double p4 = 0.0, double p5 = 0.0, double p6 = 0.0, double p7 = 0.0) -> OGRErr**

Sets the Bursa-Wolf conversion to WGS84.

This will create the TOWGS84 node as a child of the DATUM. It will fail if there's no existing DATUM node. Unlike most OGRSpatialReference methods, it will insert itself in the appropriate order, and will replace an existing TOWGS84 node if one exists.

The parameters have the same meaning as EPSG transformation 9606 (Position Vector 7-param. transformation).

- dfDX: X child in meters
- dfDY: Y child in meters
- dfDZ: Z child in meters
- dfEX: X rotation in arc seconds (optional, defaults to zero)
- dfEY: Y rotation in arc seconds (optional, defaults to zero)
- dfEZ: Z rotation in arc seconds (optional, defaults to zero)
- dfPPM: Scaling factor (parts per million)

Returns OGRERR\_NONE on success.

**SetTargetLinearUnits(string target, string name, double to\_meters) -> OGRErr**

Sets the linear units for the projection.

This method creates a UNIT subnode with the specified values as a child of the target node.

This method does the same as the C function OSRSetTargetLinearUnits().

- pszTargetKey: The keyword to set the linear units for—that is, “PROJCS” or “VERT\_CS”
- pszUnitsName: The unit's name to be used. Several preferred unit names can be found in ogr\_srs\_api.h such as SRS\_UL\_METER, SRS\_UL\_FOOT, and SRS\_UL\_US\_FOOT.
- dfInMeters: The value to multiply by a length in the indicated units to transform to meters. Several standard conversion factors can be found in ogr\_srs\_api.h.

Returns OGRERR\_NONE on success.

Since OGR 1.9.0.

**SetUTM(int zone, int north = 1) -> OGRErr**

Universal Transverse Mercator.

Sets UTM projection definition.

This will generate a projection definition with the full set of transverse mercator projection parameters for the given UTM zone. If no PROJCS[] description is set yet, one will be set to look like “UTM Zone %d, {Northern, Southern} Hemisphere.”

- nZone: UTM zone
- bNorth: TRUE for northern hemisphere or FALSE for southern hemisphere

Returns OGRERR\_NONE on success.

**SetVDG(double clong, double fe, double fn) -> OGRErr**

**SetVertCS(string VertCSName = "unnamed", string VertDatumName = "unnamed", int VertDatumType = 0) -> OGRErr**

Sets the user-visible VERT\_CS name.

This method will ensure a VERT\_CS node is created if needed. If the existing coordinate system is GEOGCS or PROJCS rooted, then it will be turned into a COMPD\_CS.

- pszVertCSName: The user-visible name of the vertical coordinate system. Not used as a key.
- pszVertDatumName: The user-visible name of the vertical datum. It’s helpful if this matches the EPSG name.
- nVertDatumType: The OGC vertical datum type, usually 2005.

Returns OGRERR\_NONE on success.

Since OGR 1.9.0.

**SetWellKnownGeogCS(string name) -> OGRErr**

Sets a GeogCS based on well-known name.

This may be called on an empty OGRSpatialReference to make a geographic coordinate system, or on something with an existing PROJCS node to set the underlying geographic coordinate system of a projected coordinate system.

The following well-known text values are currently supported:

- pszName: Name of well-known geographic coordinate system.

Returns OGRERR\_NONE on success, or OGRERR\_FAILURE if the name isn’t recognized, the target object is already initialized, or an EPSG value can’t be successfully looked up.

**StripCTParms() -> OGRErr**

Strips OGC CT Parameters.

This method will remove all components of the coordinate system that are specific to the OGC CT Specification—that is, it will attempt to strip it down to compatibility with the Simple Features 1.0 specification.

- poCurrent: Node to operate on. NULL to operate on whole tree.

Returns OGRERR\_NONE on success or an error code.

#### **Validate() -> OGRERR**

Validates SRS tokens.

This method attempts to verify that the spatial reference system is well formed, and consists of known tokens. The validation isn't comprehensive.

Returns OGRERR\_NONE if all is fine, OGRERR\_CORRUPT\_DATA if the SRS isn't well formed, and OGRERR\_UNSUPPORTED\_SRS if the SRS is well formed, but contains non-standard PROJECTION[] values.

#### **Validate() -> OGRERR**

Validates SRS tokens.

This method attempts to verify that the spatial reference system is well formed and consists of known tokens. The validation isn't comprehensive.

Returns OGRERR\_NONE if all is fine, OGRERR\_CORRUPT\_DATA if the SRS isn't well formed, and OGRERR\_UNSUPPORTED\_SRS if the SRS is well formed, but contains non-standard PROJECTION[] values.

## **D.4 Module functions**

**CreateCoordinateTransformation(SpatialReference src, SpatialReference dst) -> CoordinateTransformation**

**DontUseExceptions()**

**GetsProjectionMethods(...)**

**GetsUseExceptions() -> int**

**GetsUserInputAsWKT(char name) -> OGRERR**

**GetsWellKnownGeogCSAsWKT(char name) -> OGRERR**

**UseExceptions()**

## **D.5 Module data**

SRS\_DN\_NAD27

SRS\_DN\_NAD83

SRS\_DN\_WGS72

SRS\_DN\_WGS84

SRS\_PM\_GREENWICH



SRS\_PP\_AZIMUTH  
SRS\_PP\_CENTRAL\_MERIDIAN  
SRS\_PP\_False\_EASTING  
SRS\_PP\_False\_NORTHING  
SRS\_PP\_FIPSZONE  
SRS\_PP\_LANDSAT\_NUMBER  
SRS\_PP\_LATITUDE\_OF\_1ST\_POINT  
SRS\_PP\_LATITUDE\_OF\_2ND\_POINT  
SRS\_PP\_LATITUDE\_OF\_CENTER  
SRS\_PP\_LATITUDE\_OF\_ORIGIN  
SRS\_PP\_LATITUDE\_OF\_POINT\_1  
SRS\_PP\_LATITUDE\_OF\_POINT\_2  
SRS\_PP\_LATITUDE\_OF\_POINT\_3  
SRS\_PP\_LONGITUDE\_OF\_1ST\_POINT  
SRS\_PP\_LONGITUDE\_OF\_2ND\_POINT  
SRS\_PP\_LONGITUDE\_OF\_CENTER  
SRS\_PP\_LONGITUDE\_OF\_ORIGIN  
SRS\_PP\_LONGITUDE\_OF\_POINT\_1  
SRS\_PP\_LONGITUDE\_OF\_POINT\_2  
SRS\_PP\_LONGITUDE\_OF\_POINT\_3  
SRS\_PP\_PATH\_NUMBER  
SRS\_PP\_PERSPECTIVE\_POINT\_HEIGHT  
SRS\_PP\_PSEUDO\_STD\_PARALLEL\_1  
SRS\_PP\_RECTIFIED\_GRID\_ANGLE  
SRS\_PP\_SATELLITE\_HEIGHT  
SRS\_PP\_SCALE\_FACTOR  
SRS\_PP\_STANDARD\_PARALLEL\_1  
SRS\_PP\_STANDARD\_PARALLEL\_2  
SRS\_PP\_ZONE  
SRS\_PT\_ALBERS\_CONIC\_EQUAL\_AREA  
SRS\_PT\_AZIMUTHAL\_EQUIDISTANT  
SRS\_PT\_BONNE  
SRS\_PT\_CASSINI\_SOLDNER  
SRS\_PT\_CYLINDRICAL\_EQUAL\_AREA  
SRS\_PT\_ECKERT\_I  
SRS\_PT\_ECKERT\_II  
SRS\_PT\_ECKERT\_III  
SRS\_PT\_ECKERT\_IV  
SRS\_PT\_ECKERT\_V  
SRS\_PT\_ECKERT\_VI  
SRS\_PT\_EQUIDISTANT\_CONIC  
SRS\_PT\_EQUIRECTANGULAR

SRS\_PT\_GALL\_STEREOGRAPHIC  
SRS\_PT\_GAUSSSCHREIBERTMERCATOR  
SRS\_PT\_GEOSTATIONARY\_SATELLITE  
SRS\_PT\_GNOMONIC  
SRS\_PT\_GOODE\_HOMOLOGINE  
SRS\_PT\_HOTINE\_OBLIQUE\_MERCATOR  
SRS\_PT\_HOTINE\_OBLIQUE\_MERCATOR\_AZIMUTH\_CENTER  
SRS\_PT\_HOTINE\_OBLIQUE\_MERCATOR\_TWO\_POINT\_NATURAL\_ORIGIN  
SRS\_PT\_IGH  
SRS\_PT\_IMW\_POLYCONIC  
SRS\_PT\_KROVAK  
SRS\_PT\_LABORDE\_OBLIQUE\_MERCATOR  
SRS\_PT\_LAMBERT\_AZIMUTHAL\_EQUAL\_AREA  
SRS\_PT\_LAMBERT\_CONFORMAL\_CONIC\_1SP  
SRS\_PT\_LAMBERT\_CONFORMAL\_CONIC\_2SP  
SRS\_PT\_LAMBERT\_CONFORMAL\_CONIC\_2SP\_BELGIUM  
SRS\_PT\_MERCATOR\_1SP  
SRS\_PT\_MERCATOR\_2SP  
SRS\_PT\_MILLER\_CYLINDRICAL  
SRS\_PT\_MOLLWEIDE  
SRS\_PT\_NEW\_ZEALAND\_MAP\_GRID  
SRS\_PT\_OBLIQUE\_STEREOGRAPHIC  
SRS\_PT\_ORTHOGRAPHIC  
SRS\_PT\_POLAR\_STEREOGRAPHIC  
SRS\_PT\_POLYCONIC  
SRS\_PT\_ROBINSON  
SRS\_PT\_SINUSOIDAL  
SRS\_PT\_STEREOGRAPHIC  
SRS\_PT\_SWISS\_OBLIQUE\_CYLINDRICAL  
SRS\_PT\_TRANSVERSE\_MERCATOR  
SRS\_PT\_TRANSVERSE\_MERCATOR\_MI\_21  
SRS\_PT\_TRANSVERSE\_MERCATOR\_MI\_22  
SRS\_PT\_TRANSVERSE\_MERCATOR\_MI\_23  
SRS\_PT\_TRANSVERSE\_MERCATOR\_MI\_24  
SRS\_PT\_TRANSVERSE\_MERCATOR\_MI\_25  
SRS\_PT\_TRANSVERSE\_MERCATOR\_SOUTH\_ORIENTED  
SRS\_PT\_TUNISIA\_MINING\_GRID  
SRS\_PT\_TWO\_POINT\_EQUIDISTANT  
SRS\_PT\_VANDERGRINTEN  
SRS\_PT\_WAGNER\_I  
SRS\_PT\_WAGNER\_II  
SRS\_PT\_WAGNER\_III

SRS\_PT\_WAGNER\_IV  
SRS\_PT\_WAGNER\_V  
SRS\_PT\_WAGNER\_VI  
SRS\_PT\_WAGNER\_VII  
SRS\_UA\_DEGREE  
SRS\_UA\_DEGREE\_CONV  
SRS\_UA\_RADIAN  
SRS\_UL\_CENTIMETER  
SRS\_UL\_CENTIMETER\_CONV  
SRS\_UL\_CHAIN  
SRS\_UL\_CHAIN\_CONV  
SRS\_UL\_DECIMETER  
SRS\_UL\_DECIMETER\_CONV  
SRS\_UL\_FOOT  
SRS\_UL\_FOOT\_CONV  
SRS\_UL\_INDIAN\_CHAIN  
SRS\_UL\_INDIAN\_CHAIN\_CONV  
SRS\_UL\_INDIAN\_FOOT  
SRS\_UL\_INDIAN\_FOOT\_CONV  
SRS\_UL\_INDIAN\_YARD  
SRS\_UL\_INDIAN\_YARD\_CONV  
SRS\_UL\_INTL\_CHAIN  
SRS\_UL\_INTL\_CHAIN\_CONV  
SRS\_UL\_INTL\_FATHOM  
SRS\_UL\_INTL\_FATHOM\_CONV  
SRS\_UL\_INTL\_FOOT  
SRS\_UL\_INTL\_FOOT\_CONV  
SRS\_UL\_INTL\_INCH  
SRS\_UL\_INTL\_INCH\_CONV  
SRS\_UL\_INTL\_LINK  
SRS\_UL\_INTL\_LINK\_CONV  
SRS\_UL\_INTL\_NAUT\_MILE  
SRS\_UL\_INTL\_NAUT\_MILE\_CONV  
SRS\_UL\_INTL\_STAT\_MILE  
SRS\_UL\_INTL\_STAT\_MILE\_CONV  
SRS\_UL\_INTL\_YARD  
SRS\_UL\_INTL\_YARD\_CONV  
SRS\_UL\_KILOMETER  
SRS\_UL\_KILOMETER\_CONV  
SRS\_UL\_LINK  
SRS\_UL\_LINK\_CONV  
SRS\_UL\_LINK\_Clarke

SRS\_UL\_LINK\_Clarke\_CONV  
SRS\_UL\_METER  
SRS\_UL\_MILLIMETER  
SRS\_UL\_MILLIMETER\_CONV  
SRS\_UL\_NAUTICAL\_MILE  
SRS\_UL\_NAUTICAL\_MILE\_CONV  
SRS\_UL\_ROD  
SRS\_UL\_ROD\_CONV  
SRS\_UL\_US\_CHAIN  
SRS\_UL\_US\_CHAIN\_CONV  
SRS\_UL\_US\_FOOT  
SRS\_UL\_US\_FOOT\_CONV  
SRS\_UL\_US\_INCH  
SRS\_UL\_US\_INCH\_CONV  
SRS\_UL\_US\_STAT\_MILE  
SRS\_UL\_US\_STAT\_MILE\_CONV  
SRS\_UL\_US\_YARD  
SRS\_UL\_US\_YARD\_CONV  
SRS\_WGS84\_INVFLATTENING  
SRS\_WGS84\_SEMIMAJOR  
SRS\_WKT\_WGS84

wkbMultiPolygon  
wkbMultiPolygon25D  
wkbNDR  
wkbNone  
wkbPoint  
wkbPoint25D  
wkbPolygon  
wkbPolygon25D  
wkbUnknown  
wkbXDR



# *appendix E*

## *GDAL*

---

### **E.1 Summary**

#### **ASYNCREADER CLASS**

GetBuffer()  
GetNextUpdatedRegion()  
LockBuffer()  
UnlockBuffer()  
UnlockBuffer()

#### **BAND CLASS**

Checksum()  
ComputeBandStats()  
ComputeRasterMinMax()  
ComputeStatistics()  
CreateMaskBand()  
Fill()  
FlushCache()  
GetBand()  
GetBlockSize()  
GetCategoryNames()  
GetColorInterpretation()  
GetColorTable()  
GetDefaultHistogram()  
GetDefaultRAT()  
GetHistogram()  
GetMaskBand()  
GetMaskFlags()

GetMaximum()  
GetMinimum()  
GetNoDataValue()  
GetOffset()  
GetOverview()  
GetOverviewCount()  
GetRasterCategoryNames()  
GetRasterColorInterpretation()  
GetRasterColorTable()  
GetScale()  
GetStatistics()  
GetTiledVirtualMem()  
GetTiledVirtualMemArray()  
GetUnitType()  
GetVirtualMem()  
GetVirtualMemArray()  
GetVirtualMemAuto()  
GetVirtualMemAutoArray()  
HasArbitraryOverviews()  
ReadAsArray()  
ReadBlock()  
ReadRaster()  
ReadRaster1()  
SetCategoryNames()  
SetColorInterpretation()

SetColorTable()  
 SetDefaultHistogram()  
 SetDefaultRAT()  
 SetNoDataValue()  
 SetOffset()  
 SetRasterCategoryNames()  
 SetRasterColorInterpretation()  
 SetRasterColorTable()  
 SetScale()  
 SetStatistics()  
 SetUnitType()  
 WriteArray()  
 WriteRaster()  
 DataType  
 XSize  
 YSize

**COLORENTY CLASS**

c1  
 c2  
 c3  
 c4

**COLORTABLE CLASS**

Clone()  
 CreateColorRamp()  
 GetColorEntry()  
 GetColorEntryAsRGB()  
 GetCount()  
 GetPaletteInterpretation()  
 SetColorEntry()  
 SetColorEntry()

**DATASET CLASS**

AddBand()  
 BeginAsyncReader()  
 BuildOverviews()  
 CreateMaskBand()  
 EndAsyncReader()  
 FlushCache()  
 GetDriver()  
 GetFileList()  
 GetGCPCount()  
 GetGCPProjection()  
 GetGCPs()

GetGeoTransform()  
 GetProjection()  
 GetProjectionRef()  
 GetRasterBand()  
 GetSubDatasets()  
 GetTiledVirtualMem()  
 GetTiledVirtualMemArray()  
 GetVirtualMem()  
 GetVirtualMemArray()  
 ReadAsArray()  
 ReadRaster()  
 ReadRaster1()  
 SetGCPs()  
 SetGeoTransform()  
 SetProjection()  
 WriteRaster()  
 WriteRaster()  
 RasterCount  
 RasterXSize  
 RasterYSize

**DRIVER CLASS**

CopyFiles()  
 Create()  
 CreateCopy()  
 Delete()  
 Deregister()  
 Register()  
 Rename()  
 HelpTopic  
 LongName  
 ShortName

**GCP CLASS**

serialize()  
 GCPLine  
 GCPPixel  
 GCPX  
 GCPY  
 GCPZ  
 Id  
 Info

**MAJOROBJECT CLASS**

GetDescription()

GetMetadata()  
 GetMetadataDomainList()  
 GetMetadataItem()  
 GetMetadata\_Dict()  
 GetMetadata\_List()  
 SetDescription()  
 SetMetadata()  
 SetMetadataItem()

#### **RASTERATTRIBUTETABLE CLASS**

ChangesAreWrittenToFile()  
 Clone()  
 CreateColumn()  
 GetColOfUsage()  
 GetColumnCount()  
 GetLinearBinning()  
 GetNameOfCol()  
 GetRowCount()  
 GetRowOfValue()  
 GetTypeOfCol()  
 GetUsageOfCol()  
 GetValueAsDouble()  
 GetValueAsInt()  
 GetValueAsString()  
 ReadFromArray()  
 SetLinearBinning()  
 SetRowCount()  
 SetValueAsDouble()  
 SetValueAsInt()  
 SetValueAsString()  
 WriteArray()  
 WriteArray()

#### **STATBUF CLASS**

IsDirectory()  
 IsDirectory()  
 mode  
 mtime  
 size

#### **TRANSFORMER CLASS**

TransformGeolocations()  
 TransformPoint()  
 TransformPoints()  
 TransformPoints()

#### **VIRTUALMEM CLASS**

GetAddr()  
 Pin()  
 Pin()

#### **MODULE FUNCTIONS**

AllRegister()  
 ApplyGeoTransform()  
 AutoCreateWarpedVRT()  
 CPLBinaryToHex()  
 CPLHexToBinary()  
 ComputeMedianCutPCT()  
 ComputeProximity()  
 ContourGenerate()  
 DataTypeIsComplex()  
 Debug()  
 deprecation\_warn()  
 DecToDMS()  
 DecToPackedDMS()  
 DitherRGB2PCT()  
 DontUseExceptions()  
 Error()  
 ErrorReset()  
 EscapeString()  
 FileFromMemBuffer()  
 FillNodata()  
 FindFile()  
 FinderClean()  
 GCPsToGeoTransform()  
 GDALDestroyDriverManager()  
 GDAL\_GCP\_GCPLine\_get()  
 GDAL\_GCP\_GCPLine\_set()  
 GDAL\_GCP\_GCPPixel\_get()  
 GDAL\_GCP\_GCPPixel\_set()  
 GDAL\_GCP\_GCPX\_get()  
 GDAL\_GCP\_GCPX\_set()  
 GDAL\_GCP\_GCPY\_get()  
 GDAL\_GCP\_GCPY\_set()  
 GDAL\_GCP\_GCPZ\_get()  
 GDAL\_GCP\_GCPZ\_set()  
 GDAL\_GCP\_Id\_get()  
 GDAL\_GCP\_Id\_set()  
 GDAL\_GCP\_Info\_get()  
 GDAL\_GCP\_Info\_set()



GDAL_GCP_get_GCPLine()	Mkdir()
GDAL_GCP_get_GCPPixel()	Open()
GDAL_GCP_get_GCPX()	OpenShared()
GDAL_GCP_get_GCPY()	PackedDMSToDec()
GDAL_GCP_get_GCPZ()	ParseXMLString()
GDAL_GCP_get_Id()	Polygonize()
GDAL_GCP_get_Info()	PopErrorHandler()
GDAL_GCP_set_GCPLine()	PopFinderLocation()
GDAL_GCP_set_GCPPixel()	PushErrorHandler()
GDAL_GCP_set_GCPX()	PushFinderLocation()
GDAL_GCP_set_GCPY()	RGBFile2PCTFile()
GDAL_GCP_set_GCPZ()	RasterizeLayer()
GDAL_GCP_set_Id()	ReadDir()
GDAL_GCP_set_Info()	ReadDirRecursive()
GOA2GetAccessToken()	RegenerateOverview()
GOA2GetAuthorizationURL()	RegenerateOverviews()
GOA2GetRefreshToken()	Rename()
GeneralCmdLineProcessor()	ReprojectImage()
GetCacheMax()	Rmdir()
GetCacheUsed()	SerializeXMLTree()
GetColorInterpretationName()	SetCacheMax()
GetConfigOption()	SetConfigOption()
GetDataByTypeByName()	SetErrorHandler()
GetDataTypeName()	SieveFilter()
GetDataTypeSize()	TermProgress_nocb()
GetDriver()	Unlink()
GetDriverByName()	UseExceptions()
GetDriverCount()	VSIFCloseL()
GetLastErrorMsg()	VSIFOpenL()
GetLastErrorNo()	VSIFReadL()
GetLastErrorType()	VSIFSeekL()
GetPaletteInterpretationName()	VSIFTellL()
GetUseExceptions()	VSIFTruncateL()
HasThreadSupport()	VSIFWriteL()
IdentifyDriver()	VSIFStatL()
InvGeoTransform()	VersionInfo()

## E.2 AsyncReader class

### GetBuffer()

#### GetNextUpdatedRegion(double timeout) -> GDALAsyncStatusType

Gets async IO update.

Provides an opportunity for an asynchronous IO request to update the image buffer and return an indication of the area of the buffer that has been updated.

The dfTimeout parameter can be used to wait for additional data to become available. The timeout doesn't limit the amount of time this method may spend processing available data.

- dfTimeout: The number of seconds to wait for additional updates. Use -1 to wait indefinitely, or zero to not wait at all if no data is available.
- pnBufXOff: Location to return the X offset of the area of the request buffer that has been updated.
- pnBufYOff: Location to return the Y offset of the area of the request buffer that has been updated.
- pnBufXSize: Location to return the X size of the area of the request buffer that has been updated.
- pnBufYSize: Location to return the Y size of the area of the request buffer that has been updated.

Returns a GARIO\_ status from the following list:

- GARIO\_PENDING: No imagery was altered in the buffer, but there is still activity pending, and the application should continue to call GetNextUpdatedRegion() as time permits.
- GARIO\_UPDATE: Some of the imagery has been updated, but there is still activity pending.
- GARIO\_ERROR: Something has gone wrong. The asynchronous request should be ended.
- GARIO\_COMPLETE: An update has occurred and there is no more pending work on this request. The request should be ended and the buffer used.

#### LockBuffer(double timeout) -> int

Locks image buffer.

Locks the image buffer passed-into GDALDataset::BeginAsyncReader(). This is useful to ensure the image buffer isn't being modified while it's being used by the application. UnlockBuffer() should be used to release this lock when it's no longer needed.

- dfTimeout: The time in seconds to wait attempting to lock the buffer; -1.0 to wait indefinitely and 0 to not wait at all if it can't be acquired immediately. Default is -1.0 (infinite wait).

Returns TRUE if successful, or FALSE on an error.

**UnlockBuffer()**

Unlocks image buffer.

Releases a lock on the image buffer previously taken with LockBuffer().

**UnlockBuffer()**

Unlocks image buffer.

Releases a lock on the image buffer previously taken with LockBuffer().

**E.3 Band class**

**Checksum(int xoff = 0, int yoff = 0, int xsize = None, int ysize = None) -> int**

**ComputeBandStats(int samplestep = 1)**

**ComputeRasterMinMax(int approx\_ok = 0)**

Computes the min/max values for a band.

If approximate is OK, then the band's GetMinimum()/GetMaximum() will be trusted. If it doesn't work, a subsample of blocks will be read to get an approximate min/max. If the band has a nodata value, it will be excluded from the minimum and maximum.

If bApprox is FALSE, then all pixels will be read and used to compute an exact range.

- bApproxOK: TRUE if an approximate (faster) answer is OK; otherwise, FALSE
- adfMinMax: The array in which the minimum (adfMinMax[0]) and the maximum (adfMinMax[1]) are returned

Returns CE\_None on success or CE\_Failure on failure.

**ComputeStatistics(bool approx\_ok, GDALProgressFunc callback = 0, void callback\_data = None) -> CPLErr**

Computes image statistics.

Returns the minimum, maximum, mean, and standard deviation of all pixel values in this band. If approximate statistics are sufficient, the bApproxOK flag can be set to true in which case overviews, or a subset of image tiles may be used in computing the statistics.

Once computed, the statistics will generally be "set" back on the raster band using SetStatistics().

- bApproxOK: If TRUE, statistics may be computed based on overviews or a subset of all tiles.
- pdfMin: Location into which to load image minimum (may be NULL)
- pdfMax: Location into which to load image maximum (may be NULL)
- pdfMean: Location into which to load image mean (may be NULL)

- pdfStdDev: Location into which to load image standard deviation (may be NULL)
- pfnProgress: A function to call to report progress, or NULL
- pProgressData: Application data to pass to the progress function

Returns CE\_None on success, or CE\_Failure if an error occurs or processing is terminated by the user.

### **CreateMaskBand(int nFlags) -> CPLerr**

Adds a mask band to the current band.

The default implementation of the CreateMaskBand() method is implemented based on rules similar to the .ovr handling implemented using the GDALDefaultOverviews object. A TIFF file with the extension .msk will be created with the same base name as the original file, and it will have as many bands as the original image (or one for GMF\_PER\_DATASET). The mask images will be deflate compressed tiled images with the same block size as the original image if possible.

Note that if you got a mask band with a previous call to GetMaskBand(), it might be invalidated by CreateMaskBand(), so you have to call GetMaskBand() again.

Since GDAL 1.5.0.

Returns CE\_None on success or CE\_Failure on an error.

See also [http://trac.osgeo.org/gdal/wiki/rfc15\\_nodatabitmask](http://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask).

### **Fill(double real\_fill, double imag\_fill = 0.0) -> CPLerr**

Fills this band with a constant value.

GDAL makes no guarantees about what values pixels in newly created files are set to, so this method can be used to clear a band to a specified “default” value. The fill value is passed-in as a double but this will be converted to the underlying type before writing to the file. An optional second argument allows the imaginary component of a complex constant value to be specified.

- dfRealValue: Real component of fill value
- dfImaginaryValue: Imaginary component of fill value, defaults to zero

Returns CE\_Failure if the write fails; otherwise, CE\_None.

### **FlushCache()**

Flushes raster data cache.

This call will recover memory used to cache data blocks for this raster band, and ensure that new requests are referred to the underlying driver.

Returns CE\_None on success.

### **GetBand() -> int**

Fetches the band number.

This method returns the band that this `GDALRasterBand` object represents within its dataset. This method may return a value of zero to indicate `GDALRasterBand` objects without an apparently relationship to a dataset, such as `GDALRasterBands` serving as overviews.

Returns band number (1+) or zero if the band number isn't known.

### **GetBlockSize()**

Fetches the “natural” block size of this band.

GDAL contains a concept of the natural block size of rasters so that applications can organize data access efficiently for certain file formats. The natural block size is the block size that's most efficient for accessing the format. For many formats, this is simply a whole scan line in which case `*pnXSize` is set to `GetXSize()`, and `*pnYSize` is set to 1.

However, for tiled images this will typically be the tile size.

Note that the X and Y block sizes don't have to divide the image size evenly, meaning that right and bottom edge blocks may be incomplete. See `ReadBlock()` for an example of code dealing with these issues.

- `pnXSize`: Integer to put the X block size into or NULL.
- `pnYSize`: Integer to put the Y block size into or NULL.

### **GetCategoryNames() -> char**

Fetches the list of category names for this raster.

The return list is a “StringList” in the sense of the CPL functions. That's a NULL-terminated array of strings. Raster values without associated names will have an empty string in the returned list. The first entry in the list is for raster values of zero, and so on.

The returned string list should not be altered or freed by the application. It may change on the next GDAL call, so please copy it if it's needed for any period of time.

Returns list of names, or NULL if none.

### **GetColorInterpretation() -> GDALColorInterp**

How should this band be interpreted as color?

`GCI_Undefined` is returned when the format doesn't know anything about the color interpretation.

Returns color interpretation value for band.

### **GetColorTable() -> ColorTable**

Fetches the color table associated with band.

If it doesn't have an associated color table, the return result is NULL. The returned color table remains owned by the `GDALRasterBand`, and can't be depended on for long, nor should it ever be modified by the caller.

Returns internal color table, or NULL.

**GetDefaultHistogram(double min\_ret = None, double max\_ret = None, int buckets\_ret = None, int ppanHistogram = None, int force = 1, GDALProgressFunc callback = 0, void callback\_data = None) -> CPLerr**

Fetches default raster histogram.

The default method in GDALRasterBand will compute a default histogram. This method is overridden by derived classes (such as GDALPamRasterBand,VRTDataset, HFADataset, etc.) that could fetch efficiently an already stored histogram.

- pdfMin: Pointer to double value that will contain the lower bound of the histogram.
- pdfMax: Pointer to double value that will contain the upper bound of the histogram.
- pnBuckets: Pointer to int value that will contain the number of buckets in \*ppanHistogram.
- ppanHistogram: Pointer to array into which the histogram totals are placed. To be freed with VSIFree.
- bForce: TRUE to force the computation. If FALSE and no default histogram is available, the method will return CE\_Warning.
- pfnProgress: Function to report progress to completion.
- pProgressData: Application data to pass to pfnProgress.

Returns CE\_None on success, CE\_Failure if something goes wrong, or CE\_Warning if no default histogram is available.

**GetDefaultRAT() -> RasterAttributeTable**

Fetches default raster attribute table.

A RAT will be returned if a default one is associated with the band; otherwise, NULL is returned. The returned RAT is owned by the band and shouldn't be deleted by the application.

Returns NULL, or a pointer to an internal RAT owned by the band.

**GetHistogram(double min = -0.5, double max = 255.5, int buckets = 256, int include\_out\_of\_range = 0, int approx\_ok = 1, GDALProgressFunc callback = 0, void callback\_data = None) -> CPLerr**

Computes raster histogram.

Note that the bucket size is (dfMax-dfMin) / nBuckets.

For example, to compute a simple 256-entry histogram of 8-bit data, the following would be suitable. The unusual bounds are to ensure that bucket boundaries don't fall right on integer values causing possible errors due to rounding after scaling.

Note that setting bApproxOK will generally result in a subsampling of the file, and will utilize overviews if available. It should generally produce a representative histogram for the data that's suitable for use in generating histogram-based LUTs, for instance. Generally, bApproxOK is much faster than an exactly computed histogram.

- dfMin: The lower bound of the histogram
- dfMax: The upper bound of the histogram
- nBuckets: The number of buckets in panHistogram
- panHistogram: Array into which the histogram totals are placed
- bIncludeOutOfRange: If TRUE, values below the histogram range will be mapped into panHistogram[0], and values above will be mapped into panHistogram[nBuckets-1]; otherwise, out-of-range values are discarded
- bApproxOK: TRUE if an approximate, or incomplete histogram OK
- pfnProgress: Function to report progress to completion
- pProgressData: Application data to pass to pfnProgress

Returns CE\_None on success, or CE\_Failure if something goes wrong.

### **GetMaskBand() -> Band**

Returns the mask band associated with the band.

The GDALRasterBand class includes a default implementation of GetMaskBand() that returns one of four default implementations:

- If a corresponding .msk file exists, it will be used for the mask band.
- If the dataset has a NODATA\_VALUES metadata item, an instance of the new GDALNoDataValuesMaskBand class will be returned. GetMaskFlags() will return GMF\_NODATA | GMF\_PER\_DATASET. Since GDAL 1.6.0.
- If there is no no-data value, but the dataset has an alpha band that seems to apply to this band (specific rules yet to be determined) and that is of type GDT\_Byte, then that alpha band will be returned, and the flags GMF\_PER\_DATASET and GMF\_ALPHA will be returned in the flags.
- If neither of the above applies, an instance of the new GDALAllValidRasterBand class will be returned that has 255 values for all pixels. The null flags will return GMF\_ALL\_VALID.

Note that the GetMaskBand() should always return a GDALRasterBand mask, even if all values are 255 with the flags indicating GMF\_ALL\_VALID.

Returns a valid mask band.

Since GDAL 1.5.0.

See also [http://trac.osgeo.org/gdal/wiki/rfc15\\_nodatabitmask](http://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask).

### **GetMaskFlags() -> int**

Returns the status flags of the mask band associated with the band.

The GetMaskFlags() method returns an bitwise OR-ed set of status flags with the following available definitions that may be extended in the future:

- GMF\_ALL\_VALID(0x01): There are no invalid pixels; all mask values will be 255. When used this will normally be the only flag set.
- GMF\_PER\_DATASET(0x02): The mask band is shared between all bands on the dataset.

- **GMF\_ALPHA(0x04)**: The mask band is actually an alpha band and may have values other than 0 and 255.
- **GMF\_NODATA(0x08)**: Indicates the mask is actually being generated from no-data values. (mutually exclusive of **GMF\_ALPHA**)

The **GDALRasterBand** class includes a default implementation of **GetMaskBand()** that returns one of four default implementations:

- If a corresponding **.msk** file exists, it will be used for the mask band.
- If the dataset has a **NODATA\_VALUES** metadata item, an instance of the new **GDALNoDataValuesMaskBand** class will be returned. **GetMaskFlags()** will return **GMF\_NODATA | GMF\_PER\_DATASET**. Since GDAL 1.6.0.
- If there is no no-data value, but the dataset has an alpha band that seems to apply to this band (specific rules yet to be determined) and that is of type **GDT\_Byte**, then that alpha band will be returned, and the flags **GMF\_PER\_DATASET** and **GMF\_ALPHA** will be returned in the flags.
- If neither of the above applies, an instance of the new **GDALAllValidRasterBand** class will be returned that has 255 values for all pixels. The null flags will return **GMF\_ALL\_VALID**.

Since GDAL 1.5.0.

Returns a valid mask band.

See also [http://trac.osgeo.org/gdal/wiki/rfc15\\_nodatabitmask](http://trac.osgeo.org/gdal/wiki/rfc15_nodatabitmask).

### **GetMaximum()**

Fetches the maximum value for this band.

For file formats that don't know this intrinsically, the maximum supported value for the data type will generally be returned.

- **pbSuccess**: Pointer to a Boolean to use to indicate if the returned value is a tight maximum or not. May be **NULL** (default).

Returns the maximum raster value (excluding no data pixels).

### **GetMinimum()**

Fetches the minimum value for this band.

For file formats that don't know this intrinsically, the minimum supported value for the data type will generally be returned.

- **pbSuccess**: Pointer to a Boolean to use to indicate if the returned value is a tight minimum or not. May be **NULL** (default).

Returns the minimum raster value (excluding no data pixels).

### **GetNoDataValue()**

Fetches the no-data value for this band.

If it doesn't have out-of-data value, an out-of-range value will generally be returned. The no-data value for a band is generally a special marker value used to mark pixels



that aren't valid data. Such pixels should generally not be displayed, nor contribute to analysis operations.

- `pbSuccess`: Pointer to a Boolean to use to indicate if a value is associated with this layer. May be NULL (default).

Returns the no-data value for this band.

### **GetOffset()**

Fetches the raster value offset.

This value (in combination with the `GetScale()` value) is used to transform raw pixel values into the units returned by `GetUnits()`. For example, this might be used to store elevations in `GUInt16` bands with a precision of 0.1, and starting from -100.

Units value = (raw value \* scale) + offset

For file formats that don't know this intrinsically, a value of zero is returned.

- `pbSuccess`: Pointer to a Boolean to use to indicate if the returned value is meaningful or not. May be NULL (default).

Returns the raster offset.

### **GetOverview(int i) -> Band**

Fetches overview raster band object.

- `i`: Overview index between 0 and `GetOverviewCount()-1`.

Returns overview `GDALRasterBand`.

### **GetOverviewCount() -> int**

Returns the number of overview layers available.

Returns overview count, zero if none.

### **GetRasterCategoryNames() -> char**

### **GetRasterColorInterpretation() -> GDALColorInterp**

### **GetRasterColorTable() -> ColorTable**

### **GetScale()**

Fetches the raster value scale.

This value (in combination with the `GetOffset()` value) is used to transform raw pixel values into the units returned by `GetUnits()`. For example, this might be used to store elevations in `GUInt16` bands with a precision of 0.1, and starting from -100.

Units value = (raw value \* scale) + offset

For file formats that don't know this intrinsically, a value of one is returned.

- `pbSuccess`: Pointer to a Boolean to use to indicate if the returned value is meaningful or not. May be NULL (default).

Returns the raster scale.

### **GetStatistics(int approx\_ok, int force) -> CPLerr**

Fetches image statistics.

Returns the minimum, maximum, mean and standard deviation of all pixel values in this band. If approximate statistics are sufficient, the bApproxOK flag can be set to true, in which case overviews, or a subset of image tiles, may be used in computing the statistics.

If bForce is FALSE, results will only be returned if it can be done quickly (that is, without scanning the data). If bForce is FALSE and results cannot be returned efficiently, the method will return CE\_Warning, but no warning will have been issued. This is a nonstandard use of the CE\_Warning return value to indicate “nothing done.”

Note that file formats using PAM (persistent auxiliary metadata) services will generally cache statistics in the .pam file, allowing fast fetch after the first request.

- bApproxOK: If TRUE, statistics may be computed based on overviews or a subset of all tiles
- bForce: If FALSE, statistics will only be returned if it can be done without rescanning the image
- pdfMin: Location into which to load image minimum (may be NULL)
- pdfMax: Location into which to load image maximum (may be NULL)
- pdfMean: Location into which to load image mean (may be NULL)
- pdfStdDev: Location into which to load image standard deviation (may be NULL)

Returns CE\_None on success, CE\_Warning if no values returned, CE\_Failure if an error occurs.

### **GetTiledVirtualMem(GDALRWFlag eRWFlag, int nXOff, int nYOff, int nXSize, int nYSize, int nTileXSize, int nTileYSize, GDALDataType eBufType, size\_t nCacheSize, string options = None) -> VirtualMem**

### **GetTiledVirtualMemArray(eAccess=0, xoff=0, yoff=0, xsize=None, ysize=None, tilex-size=256, tileysize=256, datatype=None, cache\_size=10485760, options=None)**

Returns a NumPy array for the band, seen as a virtual memory mapping with a tile organization. An element is accessed with array[tiley][tilex][y][x]. Any reference to the array must be dropped before the last reference to the related dataset is also dropped.

### **GetUnitType() -> char**

Returns raster unit type.

Returns a name for the units of this raster’s values. For instance, it might be “m” for an elevation model in meters, or “ft” for feet. If no units are available, a value of ""

will be returned. The returned string should not be modified, nor freed by the calling application.

Returns unit name string.

**GetVirtualMem(GDALRWFlag eRWFlag, int nXOff, int nYOff, int nXSize, int nYSize, int nBufXSize, int nBufYSize, GDALDataType eBufType, size\_t nCacheSize, size\_t nPageSizeHint, string options = None) -> VirtualMem**

**GetVirtualMemArray(eAccess=0, xoff=0, yoff=0, xsize=None, ysize=None, bufx-size=None, bufysize=None, datatype=None, cache\_size=10485760, page\_size\_hint=0, options=None)**

Returns a NumPy array for the band, seen as a virtual memory mapping. An element is accessed with `array[y][x]`. Any reference to the array must be dropped before the last reference to the related dataset is also dropped.

**GetVirtualMemAuto(GDALRWFlag eRWFlag, string options = None) -> VirtualMem**

Creates a CPLVirtualMem object from a GDAL raster band object.

Only supported on Linux for now.

This method allows creating a virtual memory object for a GDALRasterBand that exposes the whole image data as a virtual array.

The default implementation relies on `GDALRasterBandGetVirtualMem()`, but specialized implementation, such as for raw files, may also directly use mechanisms of the operating system to create a view of the underlying file into virtual memory (`CPLVirtualMemFileMapNew()` )

At the time of writing, the GeoTIFF driver and “raw” drivers (EHdr, for example) offer a specialized implementation with direct file mapping, provided that requirements are met:

- For all drivers, the dataset must be backed by a “real” file in the file system, and the byte ordering of multibyte data types (for example, Int16) must match the native ordering of the CPU.
- In addition, for the GeoTIFF driver, the GeoTIFF file must be uncompressed and scan line-oriented (i.e., not tiled). Strips must be organized in the file in sequential order and be equally spaced (which is generally the case). Only power-of-two bit depths are supported (8 for GDT\_Byte, 16 for GDT\_Int16/GDT\_UInt16, 32 for GDT\_Float32, and 64 for GDT\_Float64)

The pointer returned remains valid until `CPLVirtualMemFree()` is called. `CPLVirtualMemFree()` must be called before the raster band object is destroyed.

If `p` is such a pointer and `base_type` the type matching `GDALGetRasterDataType()`, the element of image coordinates (`x`, `y`) can be accessed with `*(base_type*)((GByte*)p + x * *pnPixelSpace + y * *pnLineSpace)`

- `eRWFlag`: Either `GF_Reads` to read the band, or `GF_Write` to read/write the band.

- `pnPixelSpace`: Output parameter giving the byte offset from the start of one pixel value in the buffer to the start of the next pixel value within a scan line.
- `pnLineSpace`: Output parameter giving the byte offset from the start of one scan line in the buffer to the start of the next.
- `papszOptions`: NULL-terminated list of options. If a specialized implementation exists, defining `USE_DEFAULT_IMPLEMENTATION=YES` will cause the default implementation to be used. When requiring or falling back to the default implementation, the following options are available: `CACHE_SIZE` (in bytes, defaults to 40 MB), `PAGE_SIZE_HINT` (in bytes), `SINGLE_THREAD` (“FALSE” / “TRUE”, defaults to FALSE).

Returns a virtual memory object that must be unreferenced by `CPLVirtualMemFree()`, or NULL in case of failure.

Since GDAL 1.11.

#### **GetVirtualMemAutoArray(eAccess=0, options=None)**

Returns a NumPy array for the band, seen as a virtual memory mapping. An element is accessed with `array[y][x]`. Any reference to the array must be dropped before the last reference to the related dataset is also dropped.

#### **HasArbitraryOverviews() -> bool**

Checks for arbitrary overviews.

This returns TRUE if the underlying data store can compute arbitrary overviews efficiently, such as is the case with OGDIO over a network. Data stores with arbitrary overviews don’t generally have any fixed overviews, but the `RasterIO()` method can be used in downsampling mode to get overview data efficiently.

Returns TRUE if arbitrary overviews available (efficiently); otherwise, FALSE.

#### **ReadAsArray(xoff=0, yoff=0, win\_xsize=None, win\_ysize=None, buf\_xsize=None, buf\_ysize=None, buf\_obj=None)**

#### **ReadBlock(int xoff, int yoff) -> CPLerr**

Reads a block of image data efficiently.

This method accesses a “natural” block from the raster band without resampling, or data type conversion. For a more generalized but potentially less efficient access, use `RasterIO()`.

See the `GetLockedBlockRef()` method for a way of accessing internally cached block-oriented data without an extra copy into an application buffer.

- `nXBlockOff`: The horizontal block offset, with zero indicating the left-most block, 1 the next block, and so forth.
- `nYBlockOff`: The vertical block offset, with zero indicating the left-most block, 1 the next block, and so forth.

- **pImage**: The buffer into which the data will be read. The buffer must be large enough to hold `GetBlockXSize()*GetBlockYSize()` words of type `GetRasterDataType()`.

Returns `CE_None` on success or `CE_Failure` on an error.

**ReadRaster(xoff=0, yoff=0, xsize=None, ysize=None, buf\_xsize=None, buf\_ysize=None, buf\_type=None, buf\_pixel\_space=None, buf\_line\_space=None)**

**ReadRaster1(int xoff, int yoff, int xsize, int ysize, int buf\_xsize = None, int buf\_ysize = None, int buf\_type = None, int buf\_pixel\_space = None, int buf\_line\_space = None) -> CPLerr**

**SetCategoryNames(string papszCategoryNames) -> CPLerr**

Sets the category names for this band.

See the `GetCategoryNames()` method for more on the interpretation of category names.

- **papszNames**: The NULL-terminated `StringList` of category names. May be NULL to clear the existing list.

Returns `CE_None` on success or `CE_Failure` on failure. If unsupported by the driver, `CE_Failure` is returned, but no error message is reported.

**SetColorInterpretation(GDALColorInterp val) -> CPLerr**

Sets color interpretation of a band.

- **eColorInterp**: The new color interpretation to apply to this band

Returns `CE_None` on success or `CE_Failure` if method is unsupported by format.

**SetColorTable(ColorTable arg) -> int**

Sets the raster color table.

The driver will make a copy of all desired data in the color table. It remains owned by the caller after the call.

- **poCT**: The color table to apply. This may be NULL to clear the color table (where supported).

Returns `CE_None` on success or `CE_Failure` on failure. If the action is unsupported by the driver, a value of `CE_Failure` is returned, but no error is issued.

**SetDefaultHistogram(double min, double max, int buckets\_in) -> CPLerr**

Sets default histogram.

**SetDefaultRAT(RasterAttributeTable table) -> int**

Sets default raster attribute table.

Associates a default RAT with the band. If not implemented for the format, a CPLE\_NotSupported error will be issued. If successful, a copy of the RAT is made; the original remains owned by the caller.

- poRAT: The RAT to assign to the band.

Returns CE\_None on success or CE\_Failure if unsupported or otherwise failing.

#### **SetNoDataValue(double d) -> CPLErr**

Sets the no-data value for this band.

To clear the no-data value, use DeleteNoDataValue().

- dfNoData: The value to set

Returns CE\_None on success or CE\_Failure on failure. If unsupported by the driver, CE\_Failure is returned, but no error message will have been emitted.

#### **SetOffset(double val) -> CPLErr**

Sets scaling offset.

Few formats implement this method. When not implemented, it will issue a CPLE\_NotSupported error and return CE\_Failure.

- dfNewOffset: The new offset

Returns CE\_None or success or CE\_Failure on failure.

#### **SetRasterCategoryNames(string names) -> CPLErr**

#### **SetRasterColorInterpretation(GDALColorInterp val) -> CPLErr**

#### **SetRasterColorTable(ColorTable arg) -> int**

#### **SetScale(double val) -> CPLErr**

Sets scaling ratio.

Few formats implement this method. When not implemented, it will issue a CPLE\_NotSupported error and return CE\_Failure.

- dfNewScale: The new scale

Returns CE\_None or success or CE\_Failure on failure.

#### **SetStatistics(double min, double max, double mean, double stddev) -> CPLErr**

Sets statistics on band.

This method can be used to store min/max/mean/standard deviation statistics on a raster band.

The default implementation stores them as metadata, and will only work on formats that can save arbitrary metadata. This method cannot detect whether metadata will be properly saved and so may return CE\_None even if the statistics will never be saved.

- dfMin: Minimum pixel value

- dfMax: Maximum pixel value
- dfMean: Mean (average) of all pixel values
- dfStdDev: Standard deviation of all pixel values

Returns CE\_None on success or CE\_Failure on failure.

#### **SetUnitType(string val) -> CPLerr**

Sets unit type.

Sets the unit type for a raster band. Values should be one of "" (the default indicating it's unknown), "m" indicating meters, or "ft" indicating feet, though other non-standard values are allowed.

- pszNewValue: The new unit type value

Returns CE\_None on success or CE\_Failure if not successful, or unsupported.

#### **WriteArray(array, xoff=0, yoff=0)**

**WriteRaster(int xoff, int yoff, int xsize, int ysize, GIntBig buf\_len, int buf\_xsize = None, int buf\_ysize = None, int buf\_type = None, int buf\_pixel\_space = None, int buf\_line\_space = None) -> CPLerr**

#### **PROPERTIES**

##### **DataType**

Band\_DataType\_get(Band self) -> GDALDataType

##### **XSize**

Band\_XSize\_get(Band self) -> int

##### **YSize**

Band\_YSize\_get(Band self) -> int

## **E.4 ColorEntry class**

#### **PROPERTIES**

##### **c1**

ColorEntry\_c1\_get(ColorEntry self) -> short

##### **c2**

ColorEntry\_c2\_get(ColorEntry self) -> short

##### **c3**

ColorEntry\_c3\_get(ColorEntry self) -> short

##### **c4**

ColorEntry\_c4\_get(ColorEntry self) -> short

## E.5 ColorTable class

**Clone()** -> ColorTable

Makes a copy of a color table.

**CreateColorRamp(int nStartIndex, ColorEntry startcolor, int nEndIndex, ColorEntry endcolor)**

Creates color ramp.

Automatically creates a color ramp from one color entry to another. It can be called several times to create multiples ramps in the same color table.

This function is the same as the C function GDALCreateColorRamp()

- nStartIndex: Index to start the ramp on the color table [0...255]
- psStartColor: A color entry value to start the ramp
- nEndIndex: Index to end the ramp on the color table [0...255]
- psEndColor: A color entry value to end the ramp

Returns total number of entries, -1 to report error

**GetColorEntry(int entry)** -> ColorEntry

Fetches a color entry from a table.

- i: Entry offset from zero to GetColorEntryCount()-1

Returns pointer to internal color entry, or NULL if index is out of range.

**GetColorEntryAsRGB(int entry, ColorEntry centry)** -> int

Fetches a table entry in RGB format.

In theory this method should support translation of color palettes in non-RGB color spaces into RGB on the fly, but currently it only works on RGB color tables.

- i: Entry offset from zero to GetColorEntryCount()-1
- poEntry: The existing GDALColorEntry to be overwritten with the RGB values

Returns TRUE on success or FALSE if the conversion isn't supported.

**GetCount()** -> int

**GetPaletteInterpretation()** -> GDALPaletteInterp

Fetches palette interpretation.

The returned value is used to interpret the values in the GDALColorEntry.

Returns palette interpretation enumeration value, usually GPI\_RGB.

**SetColorEntry(int entry, ColorEntry centry)**

Sets entry in color table.

Note that the passed-in color entry is copied, and no internal reference to it is maintained. Also, the passed-in entry must match the color interpretation of the table to which it's being assigned.



The table is grown as needed to hold the supplied offset.

This function is the same as the C function `GDALSetColorEntry()`.

- `i`: Entry offset from zero to `GetColorEntryCount()-1`
- `poEntry`: Value to assign to table

### **SetColorEntry(int entry, ColorEntry centry)**

Sets entry in color table.

Note that the passed-in color entry is copied, and no internal reference to it is maintained. Also, the passed-in entry must match the color interpretation of the table to which it's being assigned.

The table is grown as needed to hold the supplied offset.

This function is the same as the C function `GDALSetColorEntry()`.

- `i`: Entry offset from zero to `GetColorEntryCount()-1`
- `poEntry`: Value to assign to table

## **E.6 Dataset class**

**AddBand(GDALDataType datatype = GDT\_Byte, string options = None) -> CPLerr**

**BeginAsyncReader(xoff, yoff, xsize, ysize, buf\_obj=None, buf\_xsize=None, buf\_ysize=None, buf\_type=None, band\_list=None, options=[])**

**BuildOverviews(string resampling = "NEAREST", int overviewlist = 0, GDALProgressFunc callback = 0, void callback\_data = None) -> int**

**CreateMaskBand(int nFlags) -> CPLerr**

**EndAsyncReader(AsyncReader ario)**

**FlushCache()**

**GetDriver() -> Driver**

**GetFileList() -> char**

**GetGCPCount() -> int**

**GetGCPProjection() -> char**

**GetGCPs()**

**GetGeoTransform(int can\_return\_null = None)**

**GetProjection()** -> char

**GetProjectionRef()** -> char

**GetRasterBand(int nBand)** -> Band

**GetSubDatasets()**

**GetTiledVirtualMem(GDALRWFlag eRWFlag, int nXOff, int nYOff, int nXSize, int nYSize, int nTileXSize, int nTileYSize, GDALDataType eBufType, int band\_list, GDALTileOrganization eTileOrganization, size\_t nCacheSize, string options = None)** -> VirtualMem

**GetTiledVirtualMemArray(eAccess=0, xoff=0, yoff=0, xsize=None, ysize=None, tilex-size=256, tileysize=256, datatype=None, band\_list=None, tile\_organization=2, cache\_size=10485760, options=None)**

Returns a NumPy array for the dataset, seen as a virtual memory mapping with a tile organization. If it has several bands and tile\_organization = gdal.GTO\_TIP, an element is accessed with array[tiley][tilex][y][x][band]. If it has several bands and tile\_organization = gdal.GTO\_BIT, an element is accessed with array[tiley][tilex][band][y][x]. If it has several bands and tile\_organization = gdal.GTO\_BSQ, an element is accessed with array[band][tiley][tilex][y][x]. If it has only one band, an element is accessed with array[tiley][tilex][y][x]. Any reference to the array must be dropped before the last reference to the related dataset is also dropped.

**GetVirtualMem(GDALRWFlag eRWFlag, int nXOff, int nYOff, int nXSize, int nYSize, int nBufXSize, int nBufYSize, GDALDataType eBufType, int band\_list, int bIsBand-Sequential, size\_t nCacheSize, size\_t nPageSizeHint, string options = None)** -> Virtual-Mem

**GetVirtualMemArray(eAccess=0, xoff=0, yoff=0, xsize=None, ysize=None, bufx-size=None, bufysize=None, datatype=None, band\_list=None, band\_sequential=True, cache\_size=10485760, page\_size\_hint=0, options=None)**

Returns a NumPy array for the dataset, seen as a virtual memory mapping. If it has several bands and band\_sequential = True, an element is accessed with array[band][y][x]. If it has several bands and band\_sequential = False, an element is accessed with array[y][x][band]. If it has only one band, an element is accessed with array[y][x]. Any reference to the array must be dropped before the last reference to the related dataset is also dropped.

**ReadAsArray(xoff=0, yoff=0, xsize=None, ysize=None, buf\_obj=None)**

```
ReadRaster(xoff=0, yoff=0, xsize=None, ysize=None, buf_xsize=None,
buf_ysize=None, buf_type=None, band_list=None, buf_pixel_space=None,
buf_line_space=None, buf_band_space=None)
```

```
ReadRaster1(int xoff, int yoff, int xsize, int ysize, int buf_xsize = None, int buf_ysize =
None, GDALDataType buf_type = None, int band_list = 0, int buf_pixel_space =
None, int buf_line_space = None, int buf_band_space = None) -> CPLerr
```

```
SetGCPs(int nGCPs, string pszGCPProjection) -> CPLerr
```

```
SetGeoTransform(double argin) -> CPLerr
```

```
SetProjection(string prj) -> CPLerr
```

```
WriteRaster(xoff, yoff, xsize, ysize, buf_string, buf_xsize=None, buf_ysize=None,
buf_type=None, band_list=None, buf_pixel_space=None, buf_line_space=None,
buf_band_space=None)
```

```
WriteRaster(xoff, yoff, xsize, ysize, buf_string, buf_xsize=None, buf_ysize=None,
buf_type=None, band_list=None, buf_pixel_space=None, buf_line_space=None,
buf_band_space=None)
```

#### **PROPERTIES**

##### **RasterCount**

```
Dataset_RasterCount_get(Dataset self) -> int
```

##### **RasterXSize**

```
Dataset_RasterXSize_get(Dataset self) -> int
```

##### **RasterYSize**

```
Dataset_RasterYSize_get(Dataset self) -> int
```

## **E.7 Driver class**

```
CopyFiles(string newName, string oldName) -> int
```

Copies the files of a dataset.

Copies all the files associated with a dataset.

Equivalent of the C function `GDALCopyDatasetFiles()`.

- pszNewName: New name for the dataset
- pszOldName: Old name for the dataset

Returns `CE_None` on success or `CE_Failure` if the operation fails.

**Create(string utf8\_path, int xsize, int ysize, int bands = 1, GDALDataType eType = GDT\_Byte, string options = None) -> Dataset**

Creates a new dataset with this driver.

What argument values are legal for particular drivers is driver-specific, and you have no way to query in advance to establish legal values.

That function will try to validate the creation option list passed to the driver with the GDALValidateCreationOptions() method. This check can be disabled by defining the configuration option GDAL\_VALIDATE\_CREATION\_OPTIONS=NO.

After you have finished working with the returned dataset, it's required to close it with GDALClose(). This not only closes the file handle, but also ensures that all the data and metadata has been written to the dataset (GDALFlushCache() isn't sufficient for that purpose).

In certain situations, the new dataset can be created in another process through the GDAL API Proxy mechanism.

In GDAL 2, the arguments nXSize, nYSize, and nBands can be passed to zero when creating a vector-only dataset for a compatible driver.

Equivalent of the C function GDALCreate().

- pszFilename: The name of the dataset to create. UTF-8 encoded.
- nXSize: Width of created raster in pixels.
- nYSize: Height of created raster in pixels.
- nBands: Number of bands.
- eType: Type of raster.
- papszOptions: List of driver-specific control parameters. The APPEND\_SUBDATASET=YES option can be specified to avoid prior destruction of existing dataset.

Returns NULL on failure, or a new GDALDataset.

**CreateCopy(string utf8\_path, Dataset src, int strict = 1, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> Dataset**

Creates a copy of a dataset.

This method will attempt to create a copy of a raster dataset with the indicated filename, and in this driver's format. Band number, size, type, projection, geotransform, and so forth are all to be copied from the provided template dataset.

Note that many sequential write-once formats (such as JPEG and PNG) don't implement the Create() method but do implement this CreateCopy() method. If the driver doesn't implement CreateCopy() but does implement Create(), then the default CreateCopy() mechanism built on calling Create() will be used.

It's intended that CreateCopy() will often be used with a source dataset that's a virtual dataset allowing configuration of band types, and other information without duplicating raster data (see the VRT driver). This is what's done by the gdal\_translate utility, for example.

That function will try to validate the creation option list passed to the driver with the `GDALValidateCreationOptions()` method. This check can be disabled by defining the configuration option `GDAL_VALIDATE_CREATION_OPTIONS=NO`.

After you've finished working with the returned dataset, it's required to close it with `GDALClose()`. This not only closes the file handle, but also ensures that all the data and metadata has been written to the dataset (`GDALFlushCache()` isn't sufficient for that purpose).

In certain situations, the new dataset can be created in another process through the GDAL API proxy mechanism.

- `pszFilename`: The name for the new dataset. UTF-8 encoded.
- `poSrcDS`: The dataset being duplicated.
- `bStrict`: TRUE if the copy must be strictly equivalent, or more normally FALSE indicating that the copy may adapt as needed for the output format.
- `papszOptions`: Additional format-dependent options controlling creation of the output file. The `APPEND_SUBDATASET=YES` option can be specified to avoid prior destruction of existing dataset.
- `pfnProgress`: A function to be used to report progress of the copy.
- `pProgressData`: Application data passed into progress function.

Returns a pointer to the newly created dataset (may be read-only access).

### **Delete(string utf8\_path) -> int**

Deletes named dataset.

The driver will attempt to delete the named dataset in a driver-specific fashion. Full-featured drivers will delete all associated files, database objects, or whatever is appropriate. The default behavior when no driver-specific behavior is provided is to attempt to delete the passed name as a single file.

It's unwise to have open dataset handles on this dataset when it's deleted.

Equivalent of the C function `GDALDeleteDataset()`.

- `pszFilename`: Name of dataset to delete

Returns `CE_None` on success, or `CE_Failure` if the operation fails.

### **Deregister()**

### **Register() -> int**

### **Rename(string newName, string oldName) -> int**

Renames a dataset.

Renames a dataset. This may include moving the dataset to a new directory or even a new filesystem.

It's unwise to have open dataset handles on this dataset when it is being renamed.

Equivalent of the C function `GDALRenameDataset()`.

- `pszNewName`: New name for the dataset

- pszOldName: Old name for the dataset

Returns CE\_None on success or CE\_Failure if the operation fails.

#### **4PROPERTIES**

##### **4HelpTopic**

Driver\_HelpTopic\_get(Driver self) -> char

##### **LongName**

Driver\_LongName\_get(Driver self) -> char

##### **ShortName**

Driver\_ShortName\_get(Driver self) -> char

## **E.8 GCP class**

serialize(with\_Z=0)

#### **PROPERTIES**

##### **GCPLine**

GCP\_GCPLine\_get(GCP self) -> double

##### **GCPPixel**

GCP\_GCPPixel\_get(GCP self) -> double

##### **GCPX**

GCP\_GCPX\_get(GCP self) -> double

##### **GCPY**

GCP\_GCPY\_get(GCP self) -> double

##### **GCPZ**

GCP\_GCPZ\_get(GCP self) -> double

##### **Id**

GCP\_Id\_get(GCP self) -> char

##### **Info**

GCP\_Info\_get(GCP self) -> char

## **E.9 MajorObject class**

GetDescription() -> char

Fetches object description.

The semantics of the returned description are specific to the derived type. For GDALDatasets, it's the dataset name. For GDALRasterBands, it's a description (if supported) or "".

Returns non-null pointer to the internal description string.

### **GetMetadata(domain="")**

Fetches metadata.

The returned string list is owned by the object, and may change at any time. It's formatted as a "Name=value" list with the last pointer value being NULL. Use the CPL StringList functions such as CSLFetchNameValue() to manipulate it.

Note that relatively few formats return any metadata at this time.

This method does the same thing as the C function GDALGetMetadata().

- pszDomain: The domain of interest. Use "" or NULL for the default domain.

Returns NULL or a string list.

### **GetMetadataDomainList() -> char**

Fetches list of metadata domains.

The returned string list is the list of (non-empty) metadata domains.

This method does the same thing as the C function GDALGetMetadataDomainList().

Returns NULL or a string list. Must be freed with CSLDestroy().

Since GDAL 1.11.

### **GetMetadataItem(string pszName, string pszDomain = "") -> char**

Fetches single metadata item.

The C function GDALGetMetadataItem() does the same thing as this method.

- pszName: The key for the metadata item to fetch
- pszDomain: The domain to fetch for; use NULL for the default domain

Returns NULL on failure to find the key, or a pointer to an internal copy of the value string on success.

### **GetMetadata\_Dict(string pszDomain = "") -> char**

### **GetMetadata\_List(string pszDomain = "") -> char**

### **SetDescription(string pszNewDesc)**

Sets object description.

The semantics of the description are specific to the derived type. For GDALDatasets, it's the dataset name. For GDALRasterBands, it's a description (if supported) or "".

Normally application code should not set the "description" for GDALDatasets. It's handled internally.

### **SetMetadata(string pszMetadataString, string pszDomain = "") -> CPLerr**

Sets metadata.

The C function `GDALSetMetadata()` does the same thing as this method

- `papszMetadataIn`: The metadata in name=value string list format to apply
- `pszDomain`: The domain of interest; use "" or NULL for the default domain

Returns `CE_None` on success, `CE_Failure` on failure, and `CE_Warning` if the metadata has been accepted, but is likely not maintained persistently by the underlying object between sessions.

**SetMetadataItem(string pszName, string pszValue, string pszDomain = "") -> CPLErr**

Sets single metadata item.

The C function `GDALSetMetadataItem()` does the same thing as this method.

- `pszName`: The key for the metadata item to fetch
- `pszValue`: The value to assign to the key
- `pszDomain`: The domain to set within; use NULL for the default domain

Returns `CE_None` on success, or an error code on failure.

## **E.10 RasterAttributeTable class**

**ChangesAreWrittenToFile() -> int**

Determines whether changes made to this RAT are reflected directly in the dataset.

If this returns `FALSE`, then `GDALRasterBand.SetDefaultRAT()` should be called. Otherwise, this is unnecessary because changes to this object are reflected in the dataset.

**Clone() -> RasterAttributeTable**

Copies raster attribute table.

Creates a new copy of an existing raster attribute table. The new copy becomes the responsibility of the caller to destroy. May fail (return NULL) if the attribute table is too large to clone (`GetRowCount() * GetColCount() > RAT_MAX_ELEM_FOR_CLONE`).

Returns new copy of the RAT as an in-memory implementation.

**CreateColumn(string pszName, GDALRATFieldType eType, GDALRATFieldUsage eUsage) -> int**

Creates new column.

If the table already has rows, all row values for the new column will be initialized to the default value ("" or zero). The new column is always created as the last column, and will be column (field) "`GetColumnCount()-1`" after `CreateColumn()` has completed successfully.

- `pszFieldName`: The name of the field to create
- `eFieldType`: The field type (integer, double, or string)
- `eFieldUsage`: The field usage; `GFU_Generic` if not known

Returns `CE_None` on success or `CE_Failure` if something goes wrong.



**GetColOfUsage(GDALRATFieldUsage eUsage) -> int**

Fetches column index for given usage.

Returns the index of the first column of the requested usage type, or -1 if no match is found.

- eUsage: Usage type to search for

Returns column index, or -1 on failure.

**GetColumnCount() -> int**

Fetches table column count.

Returns the number of columns.

**GetLinearBinning() -> bool**

Gets linear binning information.

Returns linear binning information if any is associated with the RAT.

- pdfRow0Min: the lower bound (pixel value) of the first category
- pdfBinSize: the width of each category (in pixel value units)

Returns TRUE if linear binning information exists or FALSE if none exists.

**GetNameOfCol(int iCol) -> char**

Fetches name of indicated column.

- iCol: The column index (zero-based)

Returns the column name or an empty string for invalid column numbers.

**GetRowCount() -> int**

Fetches row count.

Returns the number of rows.

**GetRowOfValue(double dfValue) -> int****GetTypeOfCol(int iCol) -> GDALRATFieldType**

Fetches column type.

- iCol: The column index (zero-based)

Returns column type or GFT\_Integer if the column index is illegal.

**GetUsageOfCol(int iCol) -> GDALRATFieldUsage**

Fetches column usage value.

- iCol: The column index (zero-based)

Returns the column usage, or GFU\_Generic for improper column numbers.

**GetValueAsDouble(int iRow, int iCol) -> double**

Fetches field value as a double.

The value of the requested column in the requested row is returned as a double. Non-double fields will be converted to double with the possibility of data loss.

- iRow: Row to fetch (zero-based)
- iField: Column to fetch (zero-based)

Returns field value.

**GetValueAsInt(int iRow, int iCol) -> int**

Fetches field value as an integer.

The value of the requested column in the requested row is returned as an integer. Non-integer fields will be converted to integer with the possibility of data loss.

- iRow: Row to fetch (zero-based)
- iField: Column to fetch (zero-based)

Returns field value.

**GetValueAsString(int iRow, int iCol) -> char**

Fetches field value as a string.

The value of the requested column in the requested row is returned as a string. If the field is numeric, it's formatted as a string using default rules, so some precision may be lost.

The returned string is temporary and cannot be expected to be available after the next GDAL call.

- iRow: Row to fetch (zero-based)
- iField: Column to fetch (zero-based)

Returns field value.

**ReadAsArray(field, start=0, length=None)****SetLinearBinning(double dfRow0Min, double dfBinSize) -> int**

Sets linear binning information.

For RATs with equal-sized categories (in pixel value space) that are evenly spaced, this method may be used to associate the linear binning information with the table.

- dfRow0MinIn: The lower bound (pixel value) of the first category
- dfBinSizeIn: The width of each category (in pixel value units)

Returns CE\_None on success or CE\_Failure on failure.

**SetRowCount(int nCount)**

Sets row count.

Resizes the table to include the indicated number of rows. Newly created rows will be initialized to their default values: "" for strings, and zero for numeric fields.

- `nNewCount`: The new number of rows

`SetValueAsDouble(int iRow, int iCol, double dfValue)`

`SetValueAsInt(int iRow, int iCol, int nValue)`

`SetValueAsString(int iRow, int iCol, string pszValue)`

`WriteArray(array, field, start=0)`

`WriteArray(array, field, start=0)`

### **E.11 *StatBuf class***

`IsDirectory() -> int`

`IsDirectory() -> int`

#### **PROPERTIES**

`mode`

`StatBuf_mode_get(StatBuf self) -> int`

`mtime`

`StatBuf_mtime_get(StatBuf self) -> GIntBig`

`size`

`StatBuf_size_get(StatBuf self) -> GIntBig`

### **E.12 *Transformer class***

`TransformGeolocations(Band xBand, Band yBand, Band zBand, GDALProgressFunc callback = 0, void callback_data = None, string options = None) -> int`

`TransformPoint(int bDstToSrc, double x, double y, double z = 0.0) -> int`

`TransformPoints(int bDstToSrc, int nCount) -> int`

`TransformPoints(int bDstToSrc, int nCount) -> int`

### **E.13 *VirtualMem class***

`GetAddr()`

`Pin(size_t start_offset = 0, size_t nsize = 0, int bWriteOp = 0)`

`Pin(size_t start_offset = 0, size_t nsize = 0, int bWriteOp = 0)`

**E.14 Module functions**

AllRegister()

ApplyGeoTransform(double padfGeoTransform, double dfPixel, double dfLine)

AutoCreateWarpedVRT(Dataset src\_ds, string src\_wkt = None, string dst\_wkt = None, GDALResampleAlg eResampleAlg = GRA\_NearestNeighbour, double maxerror = 0.0) -> Dataset

CPLBinaryToHex(int nBytes) -> retStringAndCPLFree

CPLHexToBinary(char pszHex, int pnBytes) -> GByte

ComputeMedianCutPCT(Band red, Band green, Band blue, int num\_colors, ColorTable colors, GDALProgressFunc callback = 0, void callback\_data = None) -> int

ComputeProximity(Band srcBand, Band proximityBand, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> int

ContourGenerate(Band srcBand, double contourInterval, double contourBase, int fixedLevelCount, int useNoData, double noDataValue, OGRLayerShadow dstLayer, int idField, int elevField, GDALProgressFunc callback = 0, void callback\_data = None) -> int

DataTypeIsComplex(GDALDataType eDataType) -> int

Debug(char msg\_class, string message)

DecToDMS(double arg0, string arg1, int arg2 = 2) -> char

DecToPackedDMS(double dfDec) -> double

deprecation\_warn(module)

DitherRGB2PCT(Band red, Band green, Band blue, Band target, ColorTable colors, GDALProgressFunc callback = 0, void callback\_data = None) -> int

DontUseExceptions()

Error(CPLErr msg\_class = CE\_Failure, int err\_code = 0, string msg = "error")

ErrorReset()

`EscapeString(int len, int scheme = CPLES_SQL) -> retStringAndCPLFree`

`FileFromMemBuffer(char utf8_path, int nBytes)`

`FillNodata(Band targetBand, Band maskBand, double maxSearchDist, int smoothing-Iterations, string options = None, GDALProgressFunc callback = 0, void callback_data = None) -> int`

`FindFile(char pszClass, string utf8_path) -> char`

`FinderClean()`

`GCPsToGeoTransform(int nGCPs, int bApproxOK = 1) -> RETURN_NONE`

`GDALDestroyDriverManager()`

`GDAL_GCP_GCPLine_get(GCP gcp) -> double`

`GDAL_GCP_GCPLine_set(GCP gcp, double dfGCPLine)`

`GDAL_GCP_GCPPixel_get(GCP gcp) -> double`

`GDAL_GCP_GCPPixel_set(GCP gcp, double dfGCPPixel)`

`GDAL_GCP_GCPX_get(GCP gcp) -> double`

`GDAL_GCP_GCPX_set(GCP gcp, double dfGCPX)`

`GDAL_GCP_GCPY_get(GCP gcp) -> double`

`GDAL_GCP_GCPY_set(GCP gcp, double dfGCPY)`

`GDAL_GCP_GCPZ_get(GCP gcp) -> double`

`GDAL_GCP_GCPZ_set(GCP gcp, double dfGCPZ)`

`GDAL_GCP_Id_get(GCP gcp) -> char`

`GDAL_GCP_Id_set(GCP gcp, string pszId)`

`GDAL_GCP_Info_get(GCP gcp) -> char`

`GDAL_GCP_Info_set(GCP gcp, string pszInfo)`

GDAL\_GCP\_get\_GCPLine(GCP gcp) -> double

GDAL\_GCP\_get\_GCPPixel(GCP gcp) -> double

GDAL\_GCP\_get\_GCPX(GCP gcp) -> double

GDAL\_GCP\_get\_GCPY(GCP gcp) -> double

GDAL\_GCP\_get\_GCPZ(GCP gcp) -> double

GDAL\_GCP\_get\_Id(GCP gcp) -> char

GDAL\_GCP\_get\_Info(GCP gcp) -> char

GDAL\_GCP\_set\_GCPLine(GCP gcp, double dfGCPLine)

GDAL\_GCP\_set\_GCPPixel(GCP gcp, double dfGCPPixel)

GDAL\_GCP\_set\_GCPX(GCP gcp, double dfGCPX)

GDAL\_GCP\_set\_GCPY(GCP gcp, double dfGCPY)

GDAL\_GCP\_set\_GCPZ(GCP gcp, double dfGCPZ)

GDAL\_GCP\_set\_Id(GCP gcp, string pszId)

GDAL\_GCP\_set\_Info(GCP gcp, string pszInfo)

GOA2GetAccessToken(char pszRefreshToken, string pszScope) -> retStringAndCPLFree

GOA2GetAuthorizationURL(char pszScope) -> retStringAndCPLFree

GOA2GetRefreshToken(char pszAuthToken, string pszScope) -> retStringAndCPLFree

GeneralCmdLineProcessor(char papszArgv, int nOptions = 0) -> char

GetCacheMax() -> GIntBig

GetCacheUsed() -> GIntBig

GetColorInterpretationName(GDALColorInterp eColorInterp) -> char

`GetConfigOption(char pszKey, string pszDefault = None) -> char`

`GetDataTypeByName(char pszDataTypeName) -> GDALDataType`

`GetDataTypeName(GDALDataType eDataType) -> char`

`GetDataTypeSize(GDALDataType eDataType) -> int`

`GetDriver(int i) -> Driver`

`GetDriverByName(char name) -> Driver`

`GetDriverCount() -> int`

`GetLastErrorMsg() -> char`

`GetLastErrorNo() -> int`

`GetLastErrorType() -> int`

`GetPaletteInterpretationName(GDALPaletteInterp ePaletteInterp) -> char`

`GetUseExceptions() -> int`

`HasThreadSupport() -> int`

`IdentifyDriver(char utf8_path, string papszSiblings = None) -> Driver`

`InvGeoTransform(double gt_in) -> int`

`Mkdir(char utf8_path, int mode) -> int`

`Open(char utf8_path, GDALAccess eAccess = GA_ReadOnly) -> Dataset`

`OpenShared(char utf8_path, GDALAccess eAccess = GA_ReadOnly) -> Dataset`

`PackedDMSToDec(double dfPacked) -> double`

`ParseXMLString(char pszXMLString) -> CPLXMLNode`

`Polygonize(Band srcBand, Band maskBand, OGRLayerShadow outLayer, int iPixValField, string options = None, GDALProgressFunc callback = 0, void callback_data = None) -> int`

PopErrorHandler()

PopFinderLocation()

PushErrorHandler(CPLEErrorHandler pfnErrorHandler = 0) -> CPLERR

PushFinderLocation(char utf8\_path)

RGBFile2PCTFile(src\_filename, dst\_filename)

RasterizeLayer(Dataset dataset, int bands, OGRLayerShadow layer, void pfnTransformer = None, void pTransformArg = None, int burn\_values = 0, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> int

ReadDir(char utf8\_path) -> char

ReadDirRecursive(char utf8\_path) -> char

RegenerateOverview(Band srcBand, Band overviewBand, string resampling = "average", GDALProgressFunc callback = 0, void callback\_data = None) -> int

RegenerateOverviews(Band srcBand, int overviewBandCount, string resampling = "average", GDALProgressFunc callback = 0, void callback\_data = None) -> int

Rename(char pszOld, string pszNew) -> int

ReprojectImage(Dataset src\_ds, Dataset dst\_ds, string src\_wkt = None, string dst\_wkt = None, GDALResampleAlg eResampleAlg = GRA\_NearestNeighbour, double WarpMemoryLimit = 0.0, double maxerror = 0.0, GDALProgressFunc callback = 0, void callback\_data = None) -> CPLERR

Rmdir(char utf8\_path) -> int

SerializeXMLTree(CPLXMLNode xmlnode) -> retStringAndCPLFree

SetCacheMax(GIntBig nBytes)

SetConfigOption(char pszKey, string pszValue)

SetErrorHandler(char pszCallbackName = None) -> CPLERR

SieveFilter(Band srcBand, Band maskBand, Band dstBand, int threshold, int connectedness = 4, string options = None, GDALProgressFunc callback = 0, void callback\_data = None) -> int



TermProgress\_noch(double dfProgress, string pszMessage = None, void pData = None) -> int

Unlink(char utf8\_path) -> int

UseExceptions()

VSIFCloseL(VSILFILE arg0)

VSIFOpenL(char utf8\_path, string pszMode) -> VSILFILE

VSIFReadL(int nMemSize, int nMemCount, VSILFILE fp) -> int

VSIFSeekL(VSILFILE arg0, GIntBig arg1, int arg2) -> int

VSIFTellL(VSILFILE arg0) -> GIntBig

VSIFTruncateL(VSILFILE arg0, GIntBig arg1) -> int

VSIFWriteL(int nLen, int size, int memb, VSILFILE f) -> int

VSISatL(char utf8\_path, int nFlags = 0) -> int

VersionInfo(char request = "VERSION\_NUM") -> char

## **E.15 Module data**

CE_Debug	CPLE_OpenFailed
CE_Failure	CPLE_OutOfMemory
CE_Fatal	CPLE_UserInterrupt
CE_None	CXT_Attribute
CE_Warning	CXT_Comment
CPLES_BackslashQuotable	CXT_Element
CPLES_CSV	CXT_Literal
CPLES_SQL	CXT_Text
CPLES_URL	DCAP_CREATE
CPLES_XML	DCAP_CREATECOPY
CPLE_AppDefined	DCAP_VIRTUALIO
CPLE_AssertionFailed	DMD_CREATIONDATATYPES
CPLE_FileIO	DMD_CREATIONOPTIONLIST
CPLE_IllegalArg	DMD_EXTENSION
CPLE_NoWriteAccess	DMD_HELPTOPIC
CPLE_None	DMD_LONGNAME
CPLE_NotSupported	DMD_MIMETYPE

DMD_SUBDATASETS	GFU_AlphaMin
GARIO_COMPLETE	GFU_Blue
GARIO_ERROR	GFU_BlueMax
GARIO_PENDING	GFU_BlueMin
GARIO_UPDATE	GFU_Generic
GA_ReadOnly	GFU_Green
GA_Update	GFU_GreenMax
GCI_AlphaBand	GFU_GreenMin
GCI_BlackBand	GFU_Max
GCI_BlueBand	GFU_MaxCount
GCI_CyanBand	GFU_Min
GCI_GrayIndex	GFU_MinMax
GCI_GreenBand	GFU_Name
GCI_HueBand	GFU_PixelCount
GCI_LightnessBand	GFU_Red
GCI_MagentaBand	GFU_RedMax
GCI_PaletteIndex	GFU_RedMin
GCI_RedBand	GF_Read
GCI_SaturationBand	GF_Write
GCI_Undefined	GMF_ALL_VALID
GCI_YCbCr_CbBand	GMF_ALPHA
GCI_YCbCr_CrBand	GMF_NODATA
GCI_YCbCr_YBand	GMF_PER_DATASET
GCI_YellowBand	GPI_CMYK
GDT_Byte	GPI_Gray
GDT_CFloat32	GPI_HLS
GDT_CFloat64	GPI_RGB
GDT_CInt16	GRA_Average
GDT_CInt32	GRA_Bilinear
GDT_Float32	GRA_Cubic
GDT_Float64	GRA_CubicSpline
GDT_Int16	GRA_Lanczos
GDT_Int32	GRA_Mode
GDT_TypeCount	GRA_NearestNeighbour
GDT_UInt16	GTO_BIT
GDT_UInt32	GTO_BSQ
GDT_Unknown	GTO_TIP
GFT_Integer	TermProgress
GFT_Real	VSI_STAT_EXISTS_FLAG
GFT_String	VSI_STAT_NATURE_FLAG
GFU_Alpha	VSI_STAT_SIZE_FLAG
GFU_AlphaMax	

# Geoprocessing with Python

Chris Garrard



**T**his book is about the science of reading, analyzing, and presenting geospatial data programmatically, using Python. Thanks to dozens of open source Python libraries and tools, you can take on professional geoprocessing tasks without investing in expensive proprietary packages like ArcGIS and MapInfo. The book shows you how.

**Geoprocessing with Python** teaches you how to access available datasets to make maps or perform your own analyses using free tools like the GDAL, NumPy, and matplotlib Python modules. Through lots of hands-on examples, you'll master core practices like handling multiple vector file formats, editing geometries, applying spatial and attribute filters, working with projections, and performing basic analyses on vector data. The book also covers how to manipulate, resample, and analyze raster data, such as aerial photographs and digital elevation models.

## What's Inside

- Geoprocessing from the ground up
- Work with vector data
- Read, write, process, and analyze raster data
- Visualize data with matplotlib
- Write custom geoprocessing tools

To read this book all you need is a basic knowledge of Python or a similar programming language.

**Chris Garrard** works as a developer for Utah State University and teaches a graduate course on Python programming for GIS.

“Marvelous resource that demonstrates the ultimate power of geospatial data processing!”

—Dr. Rizwan Bulbul  
Institute of Space Technology  
Pakistan

“Code examples may be in Python, but concepts work for all languages. Great book.”

—Karsten Strøbæk  
Microsoft, Western Europe  
Consulting Services

“A must-have for GIS professionals wishing to take their maps to the next level.”

—Jacqueline Wilson  
Cecil College

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit  
[www.manning.com/books/geoprocessing-with-python](http://www.manning.com/books/geoprocessing-with-python)

ISBN-13: 978-1-61729-214-9  
ISBN-10: 1-61729-214-1  
5 4 9 9 9



9 781617 292149