How OpenCV-Python Bindings Works?

Goal

Learn:

- How OpenCV-Python bindings are generated?
- How to extend new OpenCV modules to Python?

How OpenCV-Python bindings are generated?

In OpenCV, all algorithms are implemented in C++. But these algorithms can be used from different languages like Python, Java etc. This is made possible by the bindings generators. These generators create a bridge between C++ and Python which enables users to call C++ functions from Python. To get a complete picture of what is happening in background, a good knowledge of Python/C API is required. A simple example on extending C++ functions to Python can be found in official Python documentation[1]. So extending all functions in OpenCV to Python by writing their wrapper functions manually is a time-consuming task. So OpenCV does it in a more intelligent way. OpenCV generates these wrapper functions automatically from the C++ headers using some Python scripts which are located in `modules/python/src2`. We will look into what they do.

First, `modules/python/CMakeFiles.txt` is a CMake script which checks the modules to be extended to Python. It will automatically check all the modules to be extended and grab their header files. These header files contain list of all classes, functions, constants etc. for that particular modules.

Second, these header files are passed to a Python script, `modules/python/src2/gen2.py`. This is the Python bindings generator script. It calls another Python script `modules/python/src2/hdr_parser.py`. This is the header parser script. This header parser splits the complete header file into small Python lists. So these lists contain all details about a particular function, class etc. For example, a function will be parsed to get a list containing function name, return type, input arguments, argument types etc. Final list contains details of all the functions, enums, structs, classes etc. in that header file.

But header parser doesn't parse all the functions/classes in the header file. The developer has to specify which functions should be exported to Python. For that, there are certain macros added to the beginning of these declarations which enables the header parser to identify functions to be parsed. These macros are added by the developer who programs the particular function. In short, the developer decides which functions should be extended to Python and which are not. Details of those macros will be given in next session.

So header parser returns a final big list of parsed functions. Our generator script (gen2.py) will create wrapper functions for all the functions/classes/enums/structs parsed by header parser (You can find these header files during compilation in the `build/modules/python/` folder as pyopencv_generated_*.h files). But there may be some basic OpenCV datatypes like Mat, Vec4i, Size. They need to be extended manually. For example, a Mat type should be extended to Numpy array, Size should be extended to a tuple of two integers etc. Similarly, there may be some complex structs/classes/functions etc. which need to be extended manually. All such manual wrapper functions are placed in `modules/python/src2/cv2.cpp`.

So now only thing left is the compilation of these wrapper files which gives us **cv2** module. So when you call a function, say `res = equalizeHist(img1,img2)` in Python, you pass two numpy arrays and you expect another numpy array as the output. So these numpy arrays are converted to **cv::Mat** and then calls the **equalizeHist()** function in C++. Final result, res will be converted back into a Numpy array. So in short, almost all operations are done in C++ which gives us almost same speed as that of C++.

So this is the basic version of how OpenCV-Python bindings are generated.

Note

There is no 1:1 mapping of numpy.ndarray on **cv::Mat**. For example, **cv::Mat** has channels field, which is emulated as last dimension of numpy.ndarray and implicitly converted. However, such implicit conversion has problem with passing of 3D numpy arrays into C++ code (the last dimension is implicitly reinterpreted as number of channels). Refer to the issue for workarounds if you need to process 3D arrays or ND-arrays with channels. OpenCV 4.5.4+ has `cv.Mat` wrapper derived from `numpy.ndarray` to explicitly handle the channels behavior.

How to extend new modules to Python?

Header parser parse the header files based on some wrapper macros added to function declaration. Enumeration constants don't need any wrapper macros. They are automatically wrapped. But remaining functions, classes etc. need wrapper macros.

Functions are extended using `CV_EXPORTS_W` macro. An example is shown below.

```
CV_EXPORTS_W void equalizeHist( InputArray src, OutputArray dst );
```

Header parser can understand the input and output arguments from keywords like InputArray, OutputArray etc. But sometimes, we may need to hardcode inputs and outputs. For that, macros like `CV_OUT`, `CV_IN_OUT` etc. are used.

```
CV_EXPORTS_W void minEnclosingCircle( InputArray points,
                                      CV_OUT Point2f& center, CV_OUT float& radius );
```

For large classes also, `CV_EXPORTS_W` is used. To extend class methods, `CV_WRAP` is used. Similarly, `CV_PROP` is used for class fields.

```
class CV_EXPORTS_W CLAHE : public Algorithm
{
public:
    CV_WRAP virtual void apply(InputArray src, OutputArray dst) = 0;

    CV_WRAP virtual void setClipLimit(double clipLimit) = 0;
    CV_WRAP virtual double getClipLimit() const = 0;
}
```

Overloaded functions can be extended using `CV_EXPORTS_AS`. But we need to pass a new name so that each function will be called by that name in Python. Take the case of integral function below. Three functions are available, so each one is named with a suffix in Python. Similarly `CV_WRAP_AS` can be used to wrap overloaded methods.

```
CV_EXPORTS_W void integral( InputArray src, OutputArray sum, int sdepth = -1 );

CV_EXPORTS_AS(integral2) void integral( InputArray src, OutputArray sum,
                                        OutputArray sqsum, int sdepth = -1, int sqdepth = -1 );

CV_EXPORTS_AS(integral3) void integral( InputArray src, OutputArray sum,
                                        OutputArray sqsum, OutputArray tilted,
                                        int sdepth = -1, int sqdepth = -1 );
```

Small classes/structs are extended using `CV_EXPORTS_W_SIMPLE`. These structs are passed by value to C++ functions. Examples are `KeyPoint`, `Match` etc. Their methods are extended by `CV_WRAP` and fields are extended by `CV_PROP_RW`.

```
class CV_EXPORTS_W_SIMPLE DMatch
{
public:
    CV_WRAP DMatch();
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, float _distance);
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, int _imgIdx, float _distance);

    CV_PROP_RW int queryIdx; // query descriptor index
    CV_PROP_RW int trainIdx; // train descriptor index
    CV_PROP_RW int imgIdx;   // train image index

    CV_PROP_RW float distance;
};
```

Some other small classes/structs can be exported using `CV_EXPORTS_W_MAP` where it is exported to a Python native dictionary. `Moments()` is an example of it.

```
class CV_EXPORTS_W_MAP Moments
{
public:
    CV_PROP_RW double  m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
    CV_PROP_RW double  mu20, mu11, mu02, mu30, mu21, mu12, mu03;
    CV_PROP_RW double  nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};
```

So these are the major extension macros available in OpenCV. Typically, a developer has to put proper macros in their appropriate positions. Rest is done by generator scripts. Sometimes, there may be an exceptional cases where generator scripts cannot create the wrappers. Such functions need to be handled manually, to do this write your own `pyopencv_*.hpp` extending headers and put them into misc/python subdirectory of your module. But most of the time, a code written according to OpenCV coding guidelines will be automatically wrapped by generator scripts.

More advanced cases involves providing Python with additional features that does not exist in the C++ interface such as extra methods, type mappings, or to provide default arguments. We will take `UMat` datatype as an example of such cases later on. First, to provide Python-specific methods, `CV_WRAP_PHANTOM` is utilized in a similar manner to `CV_WRAP`, except that it takes the method header as its argument, and you would need to provide the method body in your own `pyopencv_*.hpp` extension. `UMat::queue()` and `UMat::context()` are an example of such phantom methods that does not exist in C++ interface, but are needed to handle OpenCL functionalities at the Python side. Second, if an already-existing datatype(s) is mappable to your class, it is highly preferable to indicate such capacity using `CV_WRAP_MAPPABLE` with the source type as its argument, rather than crafting your own binding function(s). This is the case of `UMat` which maps from `Mat`. Finally, if a default argument is needed, but it is not provided in the native C++ interface, you can provide it for Python side as the argument of `CV_WRAP_DEFAULT`. As per the `UMat::getMat` example below:

```
class CV_EXPORTS_W UMat
{
public:
    // You would need to provide `static bool cv_mappable_to(const Ptr<Mat>& src, Ptr<UMat>& dst)`
    CV_WRAP_MAPPABLE(Ptr<Mat>);

    /! returns the OpenCL queue used by OpenCV UMat.
    // You would need to provide the method body in the binder code
    CV_WRAP_PHANTOM(static void* queue());

    // You would need to provide the method body in the binder code
    CV_WRAP_PHANTOM(static void* context());
```

```
    CV_WRAP_AS(get) Mat getMat(int flags CV_WRAP_DEFAULT(ACCESS_RW)) const;
};
```