"""
Implementation of stream filters for PDF.

See TABLE H.1 Abbreviations for standard filter names
"""
__author__ = "Mathieu Fenniak"
__author_email__ = "biziqe@mathieu.fenniak.net"

import math
import struct
import zlib
from io import BytesIO
from typing import Any, Dict, Optional, Tuple, Union, cast

from .generic import ArrayObject, DictionaryObject, IndirectObject, NameObject

try:
    from typing import Literal # type: ignore[attr-defined]
except ImportError:
    # PEP 586 introduced typing.Literal with Python 3.8
    # For older Python versions, the backport typing_extensions is necessary:
    from typing_extensions import Literal # type: ignore[misc]

```
from ._utils import b_, deprecate_with_replacement, ord_, paeth_predictor
from .constants import CcittFaxDecodeParameters as CCITT
from .constants import ColorSpaces
from .constants import FilterTypeAbbreviations as FTA
from .constants import FilterTypes as FT
from .constants import GraphicsStateParameters as G
from .constants import ImageAttributes as IA
from .constants import LzwFilterParameters as LZW
from .constants import StreamAttributes as SA
from .errors import PdfReadError, PdfStreamError


def decompress(data: bytes) -> bytes:
    try:
        return zlib.decompress(data)
    except zlib.error:
        d = zlib.decompressobj(zlib.MAX_WBITS | 32)
        result_str = b""
        for b in [data[i : i + 1] for i in range(len(data))]:
            try:
                result_str += d.decompress(b)
            except zlib.error:
                pass
        return result_str


class FlateDecode:
    @staticmethod
    def decode(
        data: bytes,
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
        **kwargs: Any,
    ) -> bytes:
        """
        Decode data which is flate-encoded.

        :param data: flate-encoded data.
        :param decode_parms: a dictionary of values, understanding the
            "/Predictor":<int> key only
        :return: the flate-decoded data.

        :raises PdfReadError:
        """
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]
        str_data = decompress(data)
        predictor = 1
```

```
        if decode_parms:
            try:
                if isinstance(decode_parms, ArrayObject):
                    for decode_parm in decode_parms:
                        if "/Predictor" in decode_parm:
                            predictor = decode_parm["/Predictor"]
                else:
                    predictor = decode_parms.get("/Predictor", 1)
            except (AttributeError, TypeError):  # Type Error is NullObject
                pass  # Usually an array with a null object was read
        # predictor 1 == no predictor
        if predictor != 1:
            # The /Columns param. has 1 as the default value; see ISO 32000,
            # §7.4.4.3 LZWDecode and FlateDecode Parameters, Table 8
            DEFAULT_BITS_PER_COMPONENT = 8
            if isinstance(decode_parms, ArrayObject):
                columns = 1
                bits_per_component = DEFAULT_BITS_PER_COMPONENT
                for decode_parm in decode_parms:
                    if "/Columns" in decode_parm:
                        columns = decode_parm["/Columns"]
                    if LZW.BITS_PER_COMPONENT in decode_parm:
                        bits_per_component = decode_parm[LZW.BITS_PER_COMPONENT]
            else:
                columns = (
                    1 if decode_parms is None else decode_parms.get(LZW.COLUMNS, 1)
                )
                bits_per_component = (
                    decode_parms.get(LZW.BITS_PER_COMPONENT,
DEFAULT_BITS_PER_COMPONENT)
                    if decode_parms
                    else DEFAULT_BITS_PER_COMPONENT
                )

            # PNG predictor can vary by row and so is the lead byte on each row
            rowlength = (
                math.ceil(columns * bits_per_component / 8) + 1
            )  # number of bytes

            # PNG prediction:
            if 10 <= predictor <= 15:
                str_data = FlateDecode._decode_png_prediction(str_data, columns,
rowlength)  # type: ignore
            else:
                # unsupported predictor
                raise PdfReadError(f"Unsupported flatedecode predictor
{predictor!r}")
        return str_data

    @staticmethod
```

```
    def _decode_png_prediction(data: str, columns: int, rowlength: int) -> bytes:
        output = BytesIO()
        # PNG prediction can vary from row to row
        if len(data) % rowlength != 0:
            raise PdfReadError("Image data is not rectangular")
        prev_rowdata = (0,) * rowlength
        for row in range(len(data) // rowlength):
            rowdata = [
                ord_(x) for x in data[(row * rowlength) : ((row + 1) * rowlength)]
            ]
            filter_byte = rowdata[0]

            if filter_byte == 0:
                pass
            elif filter_byte == 1:
                for i in range(2, rowlength):
                    rowdata[i] = (rowdata[i] + rowdata[i - 1]) % 256
            elif filter_byte == 2:
                for i in range(1, rowlength):
                    rowdata[i] = (rowdata[i] + prev_rowdata[i]) % 256
            elif filter_byte == 3:
                for i in range(1, rowlength):
                    left = rowdata[i - 1] if i > 1 else 0
                    floor = math.floor(left + prev_rowdata[i]) / 2
                    rowdata[i] = (rowdata[i] + int(floor)) % 256
            elif filter_byte == 4:
                for i in range(1, rowlength):
                    left = rowdata[i - 1] if i > 1 else 0
                    up = prev_rowdata[i]
                    up_left = prev_rowdata[i - 1] if i > 1 else 0
                    paeth = paeth_predictor(left, up, up_left)
                    rowdata[i] = (rowdata[i] + paeth) % 256
            else:
                # unsupported PNG filter
                raise PdfReadError(f"Unsupported PNG filter {filter_byte!r}")
            prev_rowdata = tuple(rowdata)
            output.write(bytearray(rowdata[1:]))
        return output.getvalue()

    @staticmethod
    def encode(data: bytes) -> bytes:
        return zlib.compress(data)


class ASCIIHexDecode:
    """
    The ASCIIHexDecode filter decodes data that has been encoded in ASCII
    hexadecimal form into a base-7 ASCII format.
    """
```

```
    @staticmethod
    def decode(
        data: str,
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,  # noqa:
F841
        **kwargs: Any,
    ) -> str:
        """
        :param data: a str sequence of hexadecimal-encoded values to be
            converted into a base-7 ASCII string
        :param decode_parms:
        :return: a string conversion in base-7 ASCII, where each of its values
            v is such that 0 <= ord(v) <= 127.

        :raises PdfStreamError:
        """
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]  # noqa: F841
        retval = ""
        hex_pair = ""
        index = 0
        while True:
            if index >= len(data):
                raise PdfStreamError("Unexpected EOD in ASCIIHexDecode")
            char = data[index]
            if char == ">":
                break
            elif char.isspace():
                index += 1
                continue
            hex_pair += char
            if len(hex_pair) == 2:
                retval += chr(int(hex_pair, base=16))
                hex_pair = ""
            index += 1
        assert hex_pair == ""
        return retval


class LZWDecode:
    """Taken from:

http://www.java2s.com/Open-Source/Java-Document/PDF/PDF-Renderer/com/sun/pdfview/de
code/LZWDecode.java.htm
    """

    class Decoder:
        def __init__(self, data: bytes) -> None:
            self.STOP = 257
```

```
        self.CLEARDICT = 256
        self.data = data
        self.bytepos = 0
        self.bitpos = 0
        self.dict = [""] * 4096
        for i in range(256):
            self.dict[i] = chr(i)
        self.reset_dict()

    def reset_dict(self) -> None:
        self.dictlen = 258
        self.bitspercode = 9

    def next_code(self) -> int:
        fillbits = self.bitspercode
        value = 0
        while fillbits > 0:
            if self.bytepos >= len(self.data):
                return -1
            nextbits = ord_(self.data[self.bytepos])
            bitsfromhere = 8 - self.bitpos
            bitsfromhere = min(bitsfromhere, fillbits)
            value |= (
                (nextbits >> (8 - self.bitpos - bitsfromhere))
                & (0xFF >> (8 - bitsfromhere))
            ) << (fillbits - bitsfromhere)
            fillbits -= bitsfromhere
            self.bitpos += bitsfromhere
            if self.bitpos >= 8:
                self.bitpos = 0
                self.bytepos = self.bytepos + 1
        return value

    def decode(self) -> str:
        """
        TIFF 6.0 specification explains in sufficient details the steps to
        implement the LZW encode() and decode() algorithms.

        algorithm derived from:
        http://www.rasip.fer.hr/research/compress/algorithms/fund/lz/lzw.html
        and the PDFReference

        :raises PdfReadError: If the stop code is missing
        """
        cW = self.CLEARDICT
        baos = ""
        while True:
            pW = cW
            cW = self.next_code()
            if cW == -1:
```

```
                    raise PdfReadError("Missed the stop code in LZWDecode!")
                if cW == self.STOP:
                    break
                elif cW == self.CLEARDICT:
                    self.reset_dict()
                elif pW == self.CLEARDICT:
                    baos += self.dict[cW]
                else:
                    if cW < self.dictlen:
                        baos += self.dict[cW]
                        p = self.dict[pW] + self.dict[cW][0]
                        self.dict[self.dictlen] = p
                        self.dictlen += 1
                    else:
                        p = self.dict[pW] + self.dict[pW][0]
                        baos += p
                        self.dict[self.dictlen] = p
                        self.dictlen += 1
                    if (
                        self.dictlen >= (1 << self.bitspercode) - 1
                        and self.bitspercode < 12
                    ):
                        self.bitspercode += 1
            return baos

    @staticmethod
    def decode(
        data: bytes,
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
        **kwargs: Any,
    ) -> str:
        """
        :param data: ``bytes`` or ``str`` text to decode.
        :param decode_parms: a dictionary of parameter values.
        :return: decoded data.
        """
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]  # noqa: F841
        return LZWDecode.Decoder(data).decode()


class ASCII85Decode:
    """Decodes string ASCII85-encoded data into a byte format."""

    @staticmethod
    def decode(
        data: Union[str, bytes],
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
        **kwargs: Any,
```

```
    ) -> bytes:
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]  # noqa: F841
        if isinstance(data, str):
            data = data.encode("ascii")
        group_index = b = 0
        out = bytearray()
        for char in data:
            if ord("!") <= char and char <= ord("u"):
                group_index += 1
                b = b * 85 + (char - 33)
                if group_index == 5:
                    out += struct.pack(b">L", b)
                    group_index = b = 0
            elif char == ord("z"):
                assert group_index == 0
                out += b"\0\0\0\0"
            elif char == ord("~"):
                if group_index:
                    for _ in range(5 - group_index):
                        b = b * 85 + 84
                    out += struct.pack(b">L", b)[: group_index - 1]
                break
        return bytes(out)


class DCTDecode:
    @staticmethod
    def decode(
        data: bytes,
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
        **kwargs: Any,
    ) -> bytes:
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]  # noqa: F841
        return data


class JPXDecode:
    @staticmethod
    def decode(
        data: bytes,
        decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
        **kwargs: Any,
    ) -> bytes:
        if "decodeParms" in kwargs:  # pragma: no cover
            deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
            decode_parms = kwargs["decodeParms"]  # noqa: F841
```

```
        return data


class CCITParameters:
    """TABLE 3.9 Optional parameters for the CCITTFaxDecode filter."""

    def __init__(self, K: int = 0, columns: int = 0, rows: int = 0) -> None:
        self.K = K
        self.EndOfBlock = None
        self.EndOfLine = None
        self.EncodedByteAlign = None
        self.columns = columns  # width
        self.rows = rows  # height
        self.DamagedRowsBeforeError = None

    @property
    def group(self) -> int:
        if self.K < 0:
            CCITTgroup = 4
        else:
            # k == 0: Pure one-dimensional encoding (Group 3, 1-D)
            # k > 0: Mixed one- and two-dimensional encoding (Group 3, 2-D)
            CCITTgroup = 3
        return CCITTgroup


class CCITTFaxDecode:
    """
    See 3.3.5 CCITTFaxDecode Filter (PDF 1.7 Standard).

    Either Group 3 or Group 4 CCITT facsimile (fax) encoding.
    CCITT encoding is bit-oriented, not byte-oriented.

    See: TABLE 3.9 Optional parameters for the CCITTFaxDecode filter
    """

    @staticmethod
    def _get_parameters(
        parameters: Union[None, ArrayObject, DictionaryObject], rows: int
    ) -> CCITParameters:
        # TABLE 3.9 Optional parameters for the CCITTFaxDecode filter
        k = 0
        columns = 1728
        if parameters:
            if isinstance(parameters, ArrayObject):
                for decode_parm in parameters:
                    if CCITT.COLUMNS in decode_parm:
                        columns = decode_parm[CCITT.COLUMNS]
                    if CCITT.K in decode_parm:
                        k = decode_parm[CCITT.K]
```

```
        else:
            if CCITT.COLUMNS in parameters:
                columns = parameters[CCITT.COLUMNS]  # type: ignore
            if CCITT.K in parameters:
                k = parameters[CCITT.K]  # type: ignore

    return CCITParameters(k, columns, rows)

@staticmethod
def decode(
    data: bytes,
    decode_parms: Union[None, ArrayObject, DictionaryObject] = None,
    height: int = 0,
    **kwargs: Any,
) -> bytes:
    if "decodeParms" in kwargs:  # pragma: no cover
        deprecate_with_replacement("decodeParms", "parameters", "4.0.0")
        decode_parms = kwargs["decodeParms"]
    parms = CCITTFaxDecode._get_parameters(decode_parms, height)

    img_size = len(data)
    tiff_header_struct = "<2shlh" + "hhll" * 8 + "h"
    tiff_header = struct.pack(
        tiff_header_struct,
        b"II",  # Byte order indication: Little endian
        42,  # Version number (always 42)
        8,  # Offset to first IFD
        8,  # Number of tags in IFD
        256,
        4,
        1,
        parms.columns,  # ImageWidth, LONG, 1, width
        257,
        4,
        1,
        parms.rows,  # ImageLength, LONG, 1, length
        258,
        3,
        1,
        1,  # BitsPerSample, SHORT, 1, 1
        259,
        3,
        1,
        parms.group,  # Compression, SHORT, 1, 4 = CCITT Group 4 fax encoding
        262,
        3,
        1,
        0,  # Thresholding, SHORT, 1, 0 = WhiteIsZero
        273,
        4,
```

```
            1,
            struct.calcsize(
                tiff_header_struct
            ),  # StripOffsets, LONG, 1, length of header
            278,
            4,
            1,
            parms.rows,  # RowsPerStrip, LONG, 1, length
            279,
            4,
            1,
            img_size,  # StripByteCounts, LONG, 1, size of image
            0,  # last IFD
        )

        return tiff_header + data


def decode_stream_data(stream: Any) -> Union[str, bytes]:  # utils.StreamObject
    filters = stream.get(SA.FILTER, ())
    if isinstance(filters, IndirectObject):
        filters = cast(ArrayObject, filters.get_object())
    if len(filters) and not isinstance(filters[0], NameObject):
        # we have a single filter instance
        filters = (filters,)
    data: bytes = stream._data
    # If there is not data to decode we should not try to decode the data.
    if data:
        for filter_type in filters:
            if filter_type in (FT.FLATE_DECODE, FTA.FL):
                data = FlateDecode.decode(data, stream.get(SA.DECODE_PARMS))
            elif filter_type in (FT.ASCII_HEX_DECODE, FTA.AHx):
                data = ASCIIHexDecode.decode(data)  # type: ignore
            elif filter_type in (FT.LZW_DECODE, FTA.LZW):
                data = LZWDecode.decode(data, stream.get(SA.DECODE_PARMS))  # type:
ignore
            elif filter_type in (FT.ASCII_85_DECODE, FTA.A85):
                data = ASCII85Decode.decode(data)
            elif filter_type == FT.DCT_DECODE:
                data = DCTDecode.decode(data)
            elif filter_type == "/JPXDecode":
                data = JPXDecode.decode(data)
            elif filter_type == FT.CCITT_FAX_DECODE:
                height = stream.get(IA.HEIGHT, ())
                data = CCITTFaxDecode.decode(data, stream.get(SA.DECODE_PARMS),
height)
            elif filter_type == "/Crypt":
                decode_parms = stream.get(SA.DECODE_PARMS, {})
                if "/Name" not in decode_parms and "/Type" not in decode_parms:
                    pass
```

```
                else:
                    raise NotImplementedError(
                        "/Crypt filter with /Name or /Type not supported yet"
                    )
            else:
                # Unsupported filter
                raise NotImplementedError(f"unsupported filter {filter_type}")
    return data


def decodeStreamData(stream: Any) -> Union[str, bytes]:  # pragma: no cover
    deprecate_with_replacement("decodeStreamData", "decode_stream_data", "4.0.0")
    return decode_stream_data(stream)


def _xobj_to_image(x_object_obj: Dict[str, Any]) -> Tuple[Optional[str], bytes]:
    """
    Users need to have the pillow package installed.

    It's unclear if PyPDF2 will keep this function here, hence it's private.
    It might get removed at any point.

    :return: Tuple[file extension, bytes]
    """
    try:
        from PIL import Image
    except ImportError:
        raise ImportError(
            "pillow is required to do image extraction. "
            "It can be installed via 'pip install PyPDF2[image]'"
        )

    size = (x_object_obj[IA.WIDTH], x_object_obj[IA.HEIGHT])
    data = x_object_obj.get_data()  # type: ignore
    if (
        IA.COLOR_SPACE in x_object_obj
        and x_object_obj[IA.COLOR_SPACE] == ColorSpaces.DEVICE_RGB
    ):
        # https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes
        mode: Literal["RGB", "P"] = "RGB"
    else:
        mode = "P"
    extension = None
    if SA.FILTER in x_object_obj:
        if x_object_obj[SA.FILTER] == FT.FLATE_DECODE:
            extension = ".png"  # mime_type = "image/png"
            color_space = None
            if "/ColorSpace" in x_object_obj:
                color_space = x_object_obj["/ColorSpace"].get_object()
                if (
```

```
                    isinstance(color_space, ArrayObject)
                    and color_space[0] == "/Indexed"
                ):
                    color_space, base, hival, lookup = (
                        value.get_object() for value in color_space
                    )

            img = Image.frombytes(mode, size, data)
            if color_space == "/Indexed":
                from .generic import ByteStringObject

                if isinstance(lookup, ByteStringObject):
                    if base == ColorSpaces.DEVICE_GRAY and len(lookup) == hival +
1:
                        lookup = b"".join(
                            [lookup[i : i + 1] * 3 for i in range(len(lookup))]
                        )
                    img.putpalette(lookup)
                else:
                    img.putpalette(lookup.get_data())
                img = img.convert("L" if base == ColorSpaces.DEVICE_GRAY else
"RGB")
            if G.S_MASK in x_object_obj:  # add alpha channel
                alpha = Image.frombytes("L", size,
x_object_obj[G.S_MASK].get_data())
                img.putalpha(alpha)
            img_byte_arr = BytesIO()
            img.save(img_byte_arr, format="PNG")
            data = img_byte_arr.getvalue()
        elif x_object_obj[SA.FILTER] in (
            [FT.LZW_DECODE],
            [FT.ASCII_85_DECODE],
            [FT.CCITT_FAX_DECODE],
        ):
            # I'm not sure if the following logic is correct.
            # There might not be any relationship between the filters and the
            # extension
            if x_object_obj[SA.FILTER] in [[FT.LZW_DECODE], [FT.CCITT_FAX_DECODE]]:
                extension = ".tiff"  # mime_type = "image/tiff"
            else:
                extension = ".png"  # mime_type = "image/png"
            data = b_(data)
        elif x_object_obj[SA.FILTER] == FT.DCT_DECODE:
            extension = ".jpg"  # mime_type = "image/jpeg"
        elif x_object_obj[SA.FILTER] == "/JPXDecode":
            extension = ".jp2"  # mime_type = "image/x-jp2"
        elif x_object_obj[SA.FILTER] == FT.CCITT_FAX_DECODE:
            extension = ".tiff"  # mime_type = "image/tiff"
    else:
        extension = ".png"  # mime_type = "image/png"
```

```
        img = Image.frombytes(mode, size, data)
        img_byte_arr = BytesIO()
        img.save(img_byte_arr, format="PNG")
        data = img_byte_arr.getvalue()

    return extension, data
```