biip

Stein Magnus Jodal

Feb 22, 2024

USAGE

Biip is a Python library for making sense of the data in barcodes.

The library can interpret the following formats:

- GTIN-8, GTIN-12, GTIN-13, and GTIN-14 numbers, commonly found in EAN-8, EAN-13, and ITF-14 barcodes.
- GS1 AI element strings, commonly found in GS1-128 barcodes.
- UPC-A and UPC-E numbers, as found in UPC-A and UPC-E barcodes.

For details on how the barcode data is interpreted, please refer to the GS1 General Specifications (PDF).

CHAPTER

ONE

INSTALLATION

Biip requires Python 3.8 or newer.

Biip is available from PyPI:

```
$ python3 -m pip install biip
```

Optionally, with the help of `py-moneyed`, Biip can convert amounts with currency information to `moneyed.Money` objects. To install Biip with `py-moneyed`, run:

```
$ python3 -m pip install "biip[money]"
```

CHAPTER

TWO

PROJECT RESOURCES

- Documentation
- Source code
- Releases
- Issue tracker
- Contributors
- Users

2.1 Quickstart

The following examples should get you started with parsing barcode data using Biip.

See the API reference for details on the API and data fields used in the examples below.

2.1.1 Parsing barcode data

Biip's primary API is the *biip.parse()* function. It accepts a string of data from a barcode scanner and returns a *biip.ParseResult* object with any results or raises a *biip.ParseError* if all parsers fail.

Nearly all products you can buy in a store are marked with an UPC or EAN-13 barcode. These barcodes contain a number called GTIN, short for Global Trade Item Number, which can be parsed by Biip:

```
>>> import biip
>>> result = biip.parse("7032069804988")
>>> result
ParseResult(
    value='7032069804988',
    symbology_identifier=None,
    gtin=Gtin(
        value='7032069804988',
        format=GtinFormat.GTIN_13,
        prefix=GS1Prefix(value='703', usage='GS1 Norway'),
        company_prefix=GS1CompanyPrefix(value='703206'),
        payload='703206980498',
        check_digit=8,
        packaging_level=None,
    ),
```

(continues on next page)

(continued from previous page)

```
    gtin_error=None,
    upc=None,
    upc_error="Failed to parse '7032069804988' as UPC: Expected 6, 7, 8, or 12 digits,␣
↪got 13.",
    sscc=None,
    sscc_error="Failed to parse '7032069804988' as SSCC: Expected 18 digits, got 13.",
    gs1_message=GS1Message(...),
    gs1_message_error=None,
)
```

2.1.2 Error handling

Biip can parse several different data formats. Thus, it'll return a result object with a mix of results and errors. In the above example, we can see that the data is successfully parsed as a GTIN while parsing as an SSCC or GS1 Message failed, and Biip returned error messages explaining why.

If all parsers fail, Biip raises a *biip.ParseError*. The exception's string representation contains detailed error messages explaining why each parser failed to interpret the provided data:

```
>>> biip.parse("12345678")
Traceback (most recent call last):
    ...
biip._exceptions.ParseError: Failed to parse '12345678':
- Invalid GTIN check digit for '12345678': Expected 0, got 8.
- UPC: Invalid UPC-E check digit for '12345678': Expected 0, got 8.
- Failed to parse '12345678' as SSCC: Expected 18 digits, got 8.
- Failed to parse GS1 AI (12) date from '345678'.
```

Biip always checks that the GTIN check digit is correct. If the check digit doesn't match the payload, parsing fails. In this case, Biip rejected `12345678` as a GTIN-8.

2.1.3 Symbology Identifiers

If you're using a barcode scanner which has enabled Symbology Identifier prefixes, your data will have a three letter prefix, e.g. `]E0` for EAN-13 barcodes. If a Symbology Identifier is detected, Biip will detect it and only try the relevant parsers:

```
>>> biip.parse("]E09781492053743")
ParseResult(
    value=']E09781492053743',
    symbology_identifier=SymbologyIdentifier(
        value=']E0',
        symbology=Symbology.EAN_UPC,
        modifiers='0',
        gs1_symbology=GS1Symbology.EAN_13,
    ),
    gtin=Gtin(
        value='9781492053743',
        format=GtinFormat.GTIN_13,
        prefix=GS1Prefix(value='978', usage='Bookland (ISBN)'),
        company_prefix=None,
```

(continues on next page)

(continued from previous page)

```
        payload='978149205374',
        check_digit=3,
        packaging_level=None,
    ),
    gtin_error=None,
    upc=None,
    upc_error=None,
    sscc=None,
    sscc_error=None,
    gs1_message=None,
    gs1_message_error=None,
)
```

In this example, we used the ISBN from a book. As ISBNs are a subset of GTINs, this worked just like before. Because the data was prefixed by a Symbology Identifier, Biip only tried the GTIN parser. This is reflected in the lack of error messages from the SSCC and GS1 Message parsers.

2.1.4 Global Trade Item Number (GTIN)

GTINs comes in multiple formats: They are either 8, 12, 13, or 14 characters long, and the GTIN variants are accordingly named GTIN-8, GTIN-12, GTIN-13, or GTIN-14. Biip supports all GTIN formats.

Let's use the GTIN-12 `123601057072` as another example:

```
>>> import biip
>>> result = biip.parse("123601057072")
>>> result.gtin
Gtin(
    value='123601057072',
    format=GtinFormat.GTIN_12,
    prefix=GS1Prefix(value='012', usage='GS1 US'),
    company_prefix=None,
    payload='12360105707',
    check_digit=2,
    packaging_level=None,
)
```

All GTINs can be encoded as any other GTIN variant that is longer than itself. Thus, the canonical way to store a GTIN in a database is as a GTIN-14. Similarly, you'll want to convert a GTIN to GTIN-14 before using it for a database lookup:

```
>>> result.gtin.value
'123601057072'
>>> result.gtin.as_gtin_14()
'00123601057072'
```

By consistently using GTIN-14 internally in your application, you can avoid a lot of substring matching to find the database objects related to the barcode.

Restricted Circulation Number (RCN)

If you have products where the price depends on the weight of each item, and either the price or the weight are encoded in the GTIN, you are dealing with Restricted Circulation Numbers, or RCN, another subset of GTIN:

```
>>> result = biip.parse("2011122912346")
>>> result.gtin
Rcn(
    value='2011122912346',
    format=GtinFormat.GTIN_13,
    prefix=GS1Prefix(
        value='201',
        usage='Used to issue Restricted Circulation Numbers within a geographic region␣
↪(MO defined)',
    ),
    company_prefix=None,
    payload='201112291234',
    check_digit=6,
    packaging_level=None,
    usage=RcnUsage.GEOGRAPHICAL,
    region=None,
    weight=None,
    price=None,
    money=None,
)
```

In the example above, the number is detected to be an RCN, and an instance of `Rcn`, a subclass of `Gtin` with a few additional fields, is returned.

The rules for how to encode weight or price into an RCN varies between geographical regions. The national GS1 Member Organizations (MO) specify the rules for their region. Biip already supports a few of these rulesets, and you can easily add more if detailed documentation on the market's rules is available.

Because of the market variations, you must specify your geographical region for Biip to be able to extract price and weight from the RCN:

```
>>> from biip.gtin import RcnRegion
>>> result = biip.parse("2011122912346", rcn_region=RcnRegion.GREAT_BRITAIN)
>>> result.gtin
Rcn(
    value='2011122912346',
    format=GtinFormat.GTIN_13,
    prefix=GS1Prefix(
        value='201',
        usage='Used to issue Restricted Circulation Numbers within a geographic region␣
↪(MO defined)'
    ),
    company_prefix=None,
    payload='201112291234',
    check_digit=6,
    packaging_level=None,
    usage=RcnUsage.GEOGRAPHICAL,
    region=RcnRegion.GREAT_BRITAIN,
    weight=None,
    price=Decimal('12.34'),
```

(continues on next page)

(continued from previous page)

```
    money=Money('12.34', 'GBP'),
)
```

The `price` and `money` fields contain the same data. The difference is that while `price` is a simple `Decimal` type, `money` also carries currency information. The `money` field is only set if the optional dependency `py-moneyed` is installed.

2.1.5 GS1 AI Element Strings

Let us move away from consumer products.

The GS1 organization has specified a comprehensive system of Application Identifiers (AI) covering most industry use cases.

It is helpful to get the terminology straight here, as we use it throughout the Biip API:

- An *Application Identifier* (AI) is a number with 2-4 digits that specifies a data field's format and use.
- An AI prefix, together with its data field, is called an *Element String*.
- Multiple Element Strings read from a single barcode is called a *Message*.

AI Element Strings can be encoded using several different barcode types, but the linear GS1-128 barcode format is the most common.

Serial Shipping Container Code (SSCC)

If we scan a GS1-128 barcode on a pallet, we might get the data string `00157035381410375177`:

```
>>> result = biip.parse("00157035381410375177")
>>> result.gs1_message
GS1Message(
    value='00157035381410375177',
    element_strings=[
        GS1ElementString(
            ai=GS1ApplicationIdentifier(
                ai='00',
                description='Serial Shipping Container Code (SSCC)',
                data_title='SSCC',
                fnc1_required=False,
                format='N2+N18',
            ),
            value='157035381410375177',
            pattern_groups=['157035381410375177'],
            gln=None,
            gln_error=None,
            gtin=None,
            gtin_error=None,
            sscc=Sscc(
                value='157035381410375177',
                prefix=GS1Prefix(value='570', usage='GS1 Denmark'),
                company_prefix=GS1CompanyPrefix(value='5703538'),
                extension_digit=1,
                payload='15703538141037517',
                check_digit=7,
```

(continues on next page)

(continued from previous page)

```
            ),
            sscc_error=None,
            date=None,
            decimal=None,
            money=None,
        ),
    ],
)
```

From the above result, we can see that the data is a Message that contains a single Element String. The Element String has the AI `00`, which is the code for Serial Shipping Container Code, or SSCC for short.

Biip extracts the SSCC payload and validates its check digit. The result is an *Sscc* instance, with fields like `prefix` and `extension_digit`.

You can extract the Element String using *get()* and `filter()`:

```
>>> element_string = result.gs1_message.get(ai="00")
>>> element_string.ai.data_title
'SSCC'
>>> element_string.sscc.prefix.usage
'GS1 Denmark'
>>> element_string.sscc.as_hri()
'1 5703538 141037517 7'
```

In case SSCCs are what you are primarily working with, the *Sscc* instance is also available directly from *ParseResult*:

```
>>> result.sscc == element_string.sscc
True
```

If you need to display the barcode data in a more human readable way, e.g. to print below a barcode, you can use *as_hri()*:

```
>>> result.gs1_message.as_hri()
'(00)157035381410375177'
```

Product IDs, expiration dates, and lot numbers

If we unpack the pallet and scan the GS1-128 barcode on a logistic unit, containing multiple trade units, we might get the data string `0107032069804988152105261003 29`:

```
>>> result = biip.parse("0107032069804988152105261003 29")
>>> result.gs1_message.as_hri()
'(01)07032069804988(15)210526(10)0329'
```

From the human-readable interpretation (HRI) above, we can see that the data contains three Element Strings:

```
>>> result.gs1_message.element_strings
[
    GS1ElementString(
        ai=GS1ApplicationIdentifier(
            ai='01',
            description='Global Trade Item Number (GTIN)',
```

(continues on next page)

(continued from previous page)

```
            data_title='GTIN',
            fnc1_required=False,
            format='N2+N14',
        ),
        value='07032069804988',
        pattern_groups=['07032069804988'],
        gln=None,
        gln_error=None,
        gtin=Gtin(
            value='07032069804988',
            format=GtinFormat.GTIN_13,
            prefix=GS1Prefix(value='703', usage='GS1 Norway'),
            company_prefix=GS1CompanyPrefix(value='703206'),
            payload='703206980498',
            check_digit=8,
            packaging_level=None,
        ),
        gtin_error=None,
        sscc=None,
        sscc_error=None,
        date=None,
        decimal=None,
        money=None,
    ),
    GS1ElementString(
        ai=GS1ApplicationIdentifier(
            ai='15',
            description='Best before date (YYMMDD)',
            data_title='BEST BEFORE or BEST BY',
            fnc1_required=False,
            format='N2+N6',
        ),
        value='210526',
        pattern_groups=['210526'],
        gln=None,
        gln_error=None,
        gtin=None,
        gtin_error=None,
        sscc=None,
        sscc_error=None,
        date=datetime.date(2021, 5, 26),
        decimal=None,
        money=None,
    ),
    GS1ElementString(
        ai=GS1ApplicationIdentifier(
            ai='10',
            description='Batch or lot number',
            data_title='BATCH/LOT',
            fnc1_required=True,
            format='N2+X..20'
        ),
```

(continues on next page)

(continued from previous page)

```
        value='0329',
        pattern_groups=['0329'],
        gln=None,
        gln_error=None,
        gtin=None,
        gtin_error=None,
        sscc=None,
        sscc_error=None,
        date=None,
        decimal=None,
        money=None,
    ),
]
```

The first Element String is the GTIN of the trade item inside the logistic unit. As with SSCC's, this is also available directly from the *ParseResult* instance:

```
>>> result.gtin == result.gs1_message.element_strings[0].gtin
True
```

The second Element String is the expiration date of the contained trade items. To save you from interpreting the date value correctly yourself, Biip does the job for you and exposes a `date` instance:

```
>>> element_string = result.gs1_message.get(data_title="BEST BY")
>>> element_string.date
datetime.date(2021, 5, 26)
```

The last Element String is the batch or lot number of the items:

```
>>> element_string = result.gs1_message.get(ai="10")
>>> element_string.value
'0329'
```

Variable-length fields

About a third of the specified AIs don't have a fixed length. How do we then know where the Element Strings ends, and the next one starts?

In the example above, the batch/lot number, with AI `10`, is a variable-length field. You can see this from the AI format, `N2+X..20`, which indicates a two-digit AI prefix followed by a payload of up to 20 alphanumeric characters. In this case, we didn't need to do anything to handle the variable-length data field because the batch/lot number Element String was the last one in the Message.

Let's try to reorder the expiration date and batch/lot number, so that the batch/lot number comes in the middle of the Message:

```
>>> result = biip.parse("010703206980498810032915210525")
>>> result.gs1_message.as_hri()
'(01)07032069804988(10)032915210525'
```

As we can see, the batch/lot number didn't know where to stop, so it consumed the remainder of the data, including the full expiration date.

GS1-128 barcodes mark the end of variable-length Element Strings with a *Function Code 1* (FNC1) symbol. When the barcode scanner converts the barcode to a string of text, it substitutes the FNC1 symbol with something else, often with the "Group Separator" or "GS" ASCII character. The GS ASCII character has a decimal value of 29 or hexadecimal value of 0x1D.

If we insert a byte with value 0x1D, after the end of the batch/lot number, we get the following result:

```
>>> result = biip.parse("0107032069804988100329\x1d15210525")
>>> result.gs1_message.as_hri()
'(01)07032069804988(10)0329(15)210525'
```

Once again, we've correctly detected all three Element Strings.

You might need to reconfigure your barcode scanner hardware to use another separator character if:

- your barcode scanner doesn't insert the GS character, or
- some part of your scanning data pipeline cannot maintain the character as-is.

A reasonable choice for an alternative separator character might be the pipe character, `|`, as this character cannot legally be a part of the payload in Element Strings.

If we configure the barcode scanner to use an alternative separator character, we also need to tell Biip what character to expect:

```
>>> result = biip.parse("0107032069804988100329|15210525", separator_chars=["|"])
>>> result.gs1_message.as_hri()
'(01)07032069804988(10)0329(15)210525'
```

Once again, all three Element Strings was successfully extracted.

2.1.6 Deep dive

This quickstart guide covers the surface of Biip and should get you quickly up and running.

If you need to dive deeper, all parts of Biip have extensive docstrings with references to the relevant parts of specifications from GS1 and ISO. As a last resource, you have the code as well as a test suite with 100% code coverage.

Happy barcode scanning!

2.2 biip

Biip interprets the data in barcodes.

```
>>> import biip
```

An ambiguous value may be interpreted as different formats. In the following example, the value can be interpreted as either a GTIN or a GS1 Message.

```
>>> result = biip.parse("96385074")
>>> result.gtin
Gtin(value='96385074', format=GtinFormat.GTIN_8,
prefix=GS1Prefix(value='00009', usage='GS1 US'),
company_prefix=GS1CompanyPrefix(value='0000963'), payload='9638507',
check_digit=4, packaging_level=None)
```

(continues on next page)

(continued from previous page)

```
>>> result.gs1_message
GS1Message(value='96385074',
element_strings=[GS1ElementString(ai=GS1ApplicationIdentifier(ai='96',
description='Company internal information', data_title='INTERNAL',
fnc1_required=True, format='N2+X..90'), value='385074',
pattern_groups=['385074'], gln=None, gln_error=None, gtin=None,
gtin_error=None, sscc=None, sscc_error=None, date=None, decimal=None,
money=None)])
```

In the next example, the value is only valid as a GS1 Message and the GTIN parser returns an error explaining why the value cannot be interpreted as a GTIN. If a format includes check digits, Biip always control them and fail if the check digits are incorrect.

```
>>> result = biip.parse("15210527")
>>> result.gtin
None
>>> result.gtin_error
"Invalid GTIN check digit for '15210527': Expected 4, got 7."
>>> result.gs1_message
GS1Message(value='15210527',
element_strings=[GS1ElementString(ai=GS1ApplicationIdentifier(ai='15',
description='Best before date (YYMMDD)', data_title='BEST BEFORE or BEST
BY', fnc1_required=False, format='N2+N6'), value='210527',
pattern_groups=['210527'], gln=None, gln_error=None, gtin=None,
gtin_error=None, sscc=None, sscc_error=None, date=datetime.date(2021, 5,
27), decimal=None, money=None)])
```

If a value cannot be interpreted as any supported format, an exception is raised with a reason from each parser.

```
>>> biip.parse("123")
Traceback (most recent call last):
    ...
biip._exceptions.ParseError: Failed to parse '123':
- GTIN: Failed to parse '123' as GTIN: Expected 8, 12, 13, or 14 digits, got 3.
- UPC: Failed to parse '123' as UPC: Expected 6, 7, 8, or 12 digits, got 3.
- SSCC: Failed to parse '123' as SSCC: Expected 18 digits, got 3.
- GS1: Failed to match '123' with GS1 AI (12) pattern '^12(\d{6})$'.
```

biip.**parse**(*value*, ***, *rcn_region=None*, *rcn_verify_variable_measure=True*, *separator_chars=('\\x1d',)*)

Identify data format and parse data.

The current strategy is:

1. If Symbology Identifier prefix indicates a GTIN or GS1 Message, attempt to parse and validate as that.
2. Else, if not Symbology Identifier, attempt to parse with all parsers.

Parameters

- **value** (str) – The data to classify and parse.
- **rcn_region** (Optional[*RcnRegion*]) – The geographical region whose rules should be used to interpret Restricted Circulation Numbers (RCN). Needed to extract e.g. variable weight/price from GTIN.

- **rcn_verify_variable_measure** (bool) – Whether to verify that the variable measure in a RCN matches its check digit, if present. Some companies use the variable measure check digit for other purposes, requiring this check to be disabled.
- **separator_chars** (Iterable[str]) – Characters used in place of the FNC1 symbol. Defaults to <*GS*> (ASCII value 29). If variable-length fields in the middle of the message are not terminated with a separator character, the parser might greedily consume the rest of the message.

Return type
ParseResult

Returns
A data class depending upon what type of data is parsed.

Raises
ParseError – If parsing of the data fails.

class biip.**ParseResult**(*value*, *symbology_identifier=None*, *gtin=None*, *gtin_error=None*, *upc=None*, *upc_error=None*, *sscc=None*, *sscc_error=None*, *gs1_message=None*, *gs1_message_error=None*)

Results from a successful barcode parsing.

value: str
The raw value. Only stripped of surrounding whitespace.

symbology_identifier: Optional[*SymbologyIdentifier*] = None
The Symbology Identifier, if any.

gtin: Optional[*Gtin*] = None
The extracted GTIN, if any. Is also set if a GS1 Message containing a GTIN was successfully parsed.

gtin_error: Optional[str] = None
The GTIN parse error, if parsing as a GTIN was attempted and failed.

upc: Optional[*Upc*] = None
The extracted UPC, if any.

upc_error: Optional[str] = None
The UPC parse error, if parsing as an UPC was attempted and failed.

sscc: Optional[*Sscc*] = None
The extracted SSCC, if any. Is also set if a GS1 Message containing an SSCC was successfully parsed.

sscc_error: Optional[str] = None
The SSCC parse error, if parsing as an SSCC was attempted and failed.

gs1_message: Optional[*GS1Message*] = None
The extracted GS1 Message, if any.

gs1_message_error: Optional[str] = None
The GS1 Message parse error, if parsing as a GS1 Message was attempted and failed.

exception biip.**BiipException**
Base class for all custom exceptions raised by the library.

exception biip.**EncodeError**
Error raised if encoding of a value fails.

exception biip.**ParseError**

Error raised if parsing of barcode data fails.

2.3 biip.gln

Global Location Number (GLN).

GLNs are used to identify physical locations, digital locations, legal entities, and organizational functions.

If you only want to parse GLNs, you can import the GLN parser directly.

```
>>> from biip.gln import Gln
```

If parsing succeeds, it returns a *Gln* object.

```
>>> gln = Gln.parse("1234567890128")
>>> gln
Gln(value='1234567890128', prefix=GS1Prefix(value='123',
usage='GS1 US'), company_prefix=GS1CompanyPrefix(value='1234567890'),
payload='123456789012', check_digit=8)
```

As GLNs do not appear independently in barcodes, the GLN parser is not a part of the top-level parser *biip.parse()*. However, if you are parsing a barcode with GS1 element strings including a GLN, the GLN will be parsed and validated using the *Gln* class.

```
>>> import biip
>>> biip.parse("4101234567890128").gs1_message.get(data_title="SHIP TO").gln
Gln(value='1234567890128', prefix=GS1Prefix(value='123', usage='GS1 US'),
company_prefix=GS1CompanyPrefix(value='1234567890'), payload='123456789012',
check_digit=8)
```

class biip.gln.**Gln**(*value*, *prefix*, *company_prefix*, *payload*, *check_digit*)

Dataclass containing a GLN.

value: str

Raw unprocessed value.

prefix: Optional[*GS1Prefix*]

The GS1 Prefix, indicating what GS1 country organization that assigned code range.

company_prefix: Optional[*GS1CompanyPrefix*]

The GS1 Company Prefix, identifying the company that issued the GLN.

payload: str

The actual payload, including extension digit, company prefix, and item reference. Excludes the check digit.

check_digit: int

Check digit used to check if the SSCC as a whole is valid.

classmethod parse(*value*)

Parse the given value into a *Gln* object.

Parameters

value (str) – The value to parse.

Return type
Gln

Returns
GLN data structure with the successfully extracted data. The checksum is guaranteed to be valid if a GLN object is returned.

Raises
ParseError – If the parsing fails.

as_gln()
Format as a GLN.

Return type
str

2.4 biip.gs1

Support for barcode data with GS1 element strings.

The *biip.gs1* module contains Biip's support for parsing data consisting of GS1 Element Strings. Each Element String is identified by a GS1 Application Identifier (AI) prefix.

Data of this format is found in the following types of barcodes:

- GS1-128
- GS1 DataBar
- GS1 DataMatrix
- GS1 QR Code

If you only want to parse GS1 Messages, you can import the GS1 Message parser directly instead of using *biip.parse()*.

```
>>> from biip.gs1 import GS1Message
```

If the parsing succeeds, it returns a *GS1Message* object.

```
>>> msg = GS1Message.parse("0107032069804988152105261003 29")
```

The GS1Message has a raw value as well as an HRI, short for "human readable interpretation". The HRI is the text usually printed below or next to the barcode.

```
>>> msg.value
'0107032069804988152105261003 29'
>>> msg.as_hri()
'(01)07032069804988(15)210526(10)0329'
```

HRI can also be parsed.

```
>>> GS1Message.parse_hri("(01)07032069804988(15)210526(10)0329")
```

A message can contain multiple element strings.

```
>>> len(msg.element_strings)
3
```

In this example, the first element string is a GTIN.

```
>>> msg.element_strings[0]
GS1ElementString(ai=GS1ApplicationIdentifier(ai='01', description='Global
Trade Item Number (GTIN)', data_title='GTIN', fnc1_required=False,
format='N2+N14'), value='07032069804988', pattern_groups=['07032069804988'],
gln=None, gln_error=None, gtin=Gtin(value='07032069804988',
format=GtinFormat.GTIN_13, prefix=GS1Prefix(value='703', usage='GS1
Norway'), company_prefix=GS1CompanyPrefix(value='703206'),
payload='703206980498', check_digit=8, packaging_level=None),
gtin_error=None, sscc=None, sscc_error=None, date=None, decimal=None,
money=None)
```

The message object has *get()* and *filter()* methods to lookup element strings either by the Application Identifier's "data title" or its AI number.

```
>>> msg.get(data_title='BEST BY')
GS1ElementString(ai=GS1ApplicationIdentifier(ai='15', description='Best
before date (YYMMDD)', data_title='BEST BEFORE or BEST BY',
fnc1_required=False, format='N2+N6'), value='210526',
pattern_groups=['210526'], gln=None, gln_error=None, gtin=None,
gtin_error=None, sscc=None, sscc_error=None, date=datetime.date(2021, 5,
26), decimal=None, money=None)
>>> msg.get(ai="10")
GS1ElementString(ai=GS1ApplicationIdentifier(ai='10', description='Batch
or lot number', data_title='BATCH/LOT', fnc1_required=True,
format='N2+X..20'), value='0329', pattern_groups=['0329'], gln=None,
gln_error=None, gtin=None, gtin_error=None, sscc=None, sscc_error=None,
date=None, decimal=None, money=None)
```

class biip.gs1.**GS1CompanyPrefix**(*value*)

Company prefix assigned by GS1.

The prefix assigned to a single company.

Example

```
>>> from biip.gs1 import GS1CompanyPrefix
>>> GS1CompanyPrefix.extract("7044610873466")
GS1CompanyPrefix(value='704461')
```

value: str

The company prefix itself.

classmethod extract(*value*)

Extract the GS1 Company Prefix from the given value.

Parameters

value (str) – The string to extract a GS1 Company Prefix from. The value is typically a GLN, GTIN, or an SSCC.

Return type

Optional[*GS1CompanyPrefix*]

Returns
Metadata about the extracted prefix, or *None* if the prefix is unknown.

Raises
ParseError – If the parsing fails.

class biip.gs1.**GS1Message**(*value*, *element_strings*)

A GS1 message is the result of a single barcode scan.

It may contain one or more GS1 Element Strings.

Example

See *biip.gs1* for a usage example.

value: str

Raw unprocessed value.

element_strings: List[*GS1ElementString*]

List of Element Strings found in the message.

classmethod parse(*value*, *, *rcn_region=None*, *rcn_verify_variable_measure=True*, *separator_chars=('\\x1d',)*)

Parse a string from a barcode scan as a GS1 message with AIs.

Parameters

- **value** (str) – The string to parse.
- **rcn_region** (Optional[*RcnRegion*]) – The geographical region whose rules should be used to interpret Restricted Circulation Numbers (RCN). Needed to extract e.g. variable weight/price from GTIN.
- **rcn_verify_variable_measure** (bool) – Whether to verify that the variable measure in a RCN matches its check digit, if present. Some companies use the variable measure check digit for other purposes, requiring this check to be disabled.
- **separator_chars** (Iterable[str]) – Characters used in place of the FNC1 symbol. Defaults to <*GS*> (ASCII value 29). If variable-length fields in the middle of the message are not terminated with a separator character, the parser might greedily consume the rest of the message.

Return type
GS1Message

Returns
A message object with one or more element strings.

Raises
ParseError – If a fixed-length field ends with a separator character.

classmethod parse_hri(*value*, *, *rcn_region=None*, *rcn_verify_variable_measure=True*)

Parse the GS1 string given in HRI (human readable interpretation) format.

Parameters

- **value** (str) – The HRI string to parse.
- **rcn_region** (Optional[*RcnRegion*]) – The geographical region whose rules should be used to interpret Restricted Circulation Numbers (RCN). Needed to extract e.g. variable weight/price from GTIN.

- **rcn_verify_variable_measure** (bool) – Whether to verify that the variable measure in a RCN matches its check digit, if present. Some companies use the variable measure check digit for other purposes, requiring this check to be disabled.

Return type
GS1Message

Returns
A message object with one or more element strings.

Raises
ParseError – If parsing of the data fails.

as_hri()

Render as a human readable interpretation (HRI).

The HRI is often printed directly below barcodes.

Return type
str

Returns
A human-readable string where the AIs are wrapped in parenthesis.

filter(*, *ai=None*, *data_title=None*)

Filter Element Strings by AI or data title.

Parameters

- **ai** (Union[str, *GS1ApplicationIdentifier*, None]) – AI instance or string to match against the start of the Element String's AI.
- **data_title** (Optional[str]) – String to find anywhere in the Element String's AI data title.

Return type
List[*GS1ElementString*]

Returns
All matching Element Strings in the message.

get(*, *ai=None*, *data_title=None*)

Get Element String by AI or data title.

Parameters

- **ai** (Union[str, *GS1ApplicationIdentifier*, None]) – AI instance or string to match against the start of the Element String's AI.
- **data_title** (Optional[str]) – String to find anywhere in the Element String's AI data title.

Return type
Optional[*GS1ElementString*]

Returns
The first matching Element String in the message.

class biip.gs1.**GS1ElementString**(*ai*, *value*, *pattern_groups*, *gln=None*, *gln_error=None*, *gtin=None*, *gtin_error=None*, *sscc=None*, *sscc_error=None*, *date=None*, *decimal=None*, *money=None*)

GS1 Element String.

An Element String consists of a GS1 Application Identifier (AI) and its data field.

A single barcode can contain multiple Element Strings. Together these are called a "message."

Example

```
>>> from biip.gs1 import GS1ElementString
>>> element_string = GS1ElementString.extract("0107032069804988")
>>> element_string
GS1ElementString(ai=GS1ApplicationIdentifier(ai='01',
description='Global Trade Item Number (GTIN)', data_title='GTIN',
fnc1_required=False, format='N2+N14'), value='07032069804988',
pattern_groups=['07032069804988'], gln=None, gln_error=None,
gtin=Gtin(value='07032069804988', format=GtinFormat.GTIN_13,
prefix=GS1Prefix(value='703', usage='GS1 Norway'),
company_prefix=GS1CompanyPrefix(value='703206'), payload='703206980498',
check_digit=8, packaging_level=None), gtin_error=None, sscc=None,
sscc_error=None, date=None, decimal=None, money=None)
>>> element_string.as_hri()
'(01)07032069804988'
```

ai: *GS1ApplicationIdentifier*

The element's Application Identifier (AI).

value: **str**

Raw data field of the Element String. Does not include the AI.

pattern_groups: **List[str]**

List of pattern groups extracted from the Element String.

gln: **Optional[*Gln*] = None**

A GLN created from the element string, if the AI represents a GLN.

gln_error: **Optional[str] = None**

The GLN parse error, if parsing as a GLN was attempted and failed.

gtin: **Optional[*Gtin*] = None**

A GTIN created from the element string, if the AI represents a GTIN.

gtin_error: **Optional[str] = None**

The GTIN parse error, if parsing as a GTIN was attempted and failed.

sscc: **Optional[*Sscc*] = None**

An SSCC created from the element string, if the AI represents a SSCC.

sscc_error: **Optional[str] = None**

The SSCC parse error, if parsing as an SSCC was attempted and failed.

date: **Optional[date] = None**

A date created from the element string, if the AI represents a date.

decimal: **Optional[Decimal] = None**

A decimal value created from the element string, if the AI represents a number.

money: Optional[Money] = None

A Money value created from the element string, if the AI represents a currency and an amount. Only set if py-moneyed is installed.

classmethod extract(*value*, ***, *rcn_region=None*, *rcn_verify_variable_measure=True*, *separator_chars=('\\x1d',)*)

Extract the first GS1 Element String from the given value.

If the element string contains a primitive data type, like a date, decimal number, or currency, it will be parsed and stored in the `date`, `decimal`, or `money` field respectively. If parsing of a primitive data type fails, a `ParseError` will be raised.

If the element string contains another supported format, like a GLN, GTIN, or SSCC, it will parsed and validated, and the result stored in the fields `gln`, `gtin`, or `sscc` respectively. If parsing or validation of an inner format fails, the `gln_error`, `gtin_error`, or `sscc_error` field will be set. No `ParseError` will be raised.

Parameters

- **value** (`str`) – The string to extract an Element String from. May contain more than one Element String.
- **rcn_region** (`Optional[RcnRegion]`) – The geographical region whose rules should be used to interpret Restricted Circulation Numbers (RCN). Needed to extract e.g. variable weight/price from GTIN.
- **rcn_verify_variable_measure** (`bool`) – Whether to verify that the variable measure in a RCN matches its check digit, if present. Some companies use the variable measure check digit for other purposes, requiring this check to be disabled.
- **separator_chars** (`Iterable[str]`) – Characters used in place of the FNC1 symbol. Defaults to <*GS*> (ASCII value 29). If variable-length fields are not terminated with a separator character, the parser might greedily consume later fields.

Return type
GS1ElementString

Returns
A data class with the Element String's parts and data extracted from it.

Raises

- **ValueError** – If the `separator_char` isn't exactly 1 character long.
- ***ParseError*** – If the parsing fails.

as_hri()

Render as a human readable interpretation (HRI).

The HRI is often printed directly below the barcode.

Return type
`str`

Returns
A human-readable string where the AI is wrapped in parenthesis.

class biip.gs1.**GS1ApplicationIdentifier**(*ai*, *description*, *data_title*, *fnc1_required*, *format*, *pattern*)

GS1 Application Identifier (AI).

AIs are data field prefixes used in several types of barcodes, including GS1-128. The AI defines what semantical meaning and format of the following data field.

AIs standardize how to include e.g. product weight, expiration dates, and lot numbers in barcodes.

References

https://www.gs1.org/standards/barcodes/application-identifiers

Example

```
>>> from biip.gs1 import GS1ApplicationIdentifier
>>> ai = GS1ApplicationIdentifier.extract("01")
>>> ai
GS1ApplicationIdentifier(ai='01', description='Global Trade Item
Number (GTIN)', data_title='GTIN', fnc1_required=False,
format='N2+N14')
>>> ai.pattern
'^01(\\d{14})$'
```

ai: str

The Application Identifier (AI) itself.

description: str

Description of the AIs use.

data_title: str

Commonly used label/abbreviation for the AI.

fnc1_required: bool

Whether a FNC1 character is required after element strings of this type. This is the case for all AIs that have a variable length.

format: str

Human readable format of the AIs element string.

pattern: str

Regular expression for parsing the AIs element string.

classmethod extract(*value*)

Extract the GS1 Application Identifier (AI) from the given value.

Parameters

value (str) – The string to extract an AI from.

Return type

GS1ApplicationIdentifier

Returns

Metadata about the extracted AI.

Raises

ParseError – If the parsing fails.

Example

```
>>> from biip.gs1 import GS1ApplicationIdentifier
>>> GS1ApplicationIdentifier.extract("0107032069804988152105261003 29")
GS1ApplicationIdentifier(ai='01', description='Global Trade Item
Number (GTIN)', data_title='GTIN', fnc1_required=False,
format='N2+N14')
```

class biip.gs1.**GS1Prefix**(*value*, *usage*)

Prefix assigned by GS1.

Used to split the allocation space of various number schemes, e.g. GTIN, among GS1 organizations worldwide.

The GS1 Prefix does not identify the origin of a product, only where the number was assigned to a GS1 member organization.

References

https://www.gs1.org/standards/id-keys/company-prefix

Example

```
>>> from biip.gs1 import GS1Prefix
>>> GS1Prefix.extract("978-1-492-05374-3")
GS1Prefix(value='978', usage='Bookland (ISBN)')
```

value: str

The prefix itself.

usage: str

Description of who is using the prefix.

classmethod extract(*value*)

Extract the GS1 Prefix from the given value.

Parameters
value (str) – The string to extract a GS1 Prefix from.

Return type
Optional[*GS1Prefix*]

Returns
Metadata about the extracted prefix, or *None* if the prefix is unknown.

Raises
ParseError – If the parsing fails.

class biip.gs1.**GS1Symbology**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of Symbology Identifiers used in the GS1 system.

References

GS1 General Specifications, figure 5.1.2-2. ISO/IEC 15424:2008.

EAN_13 = 'E0'

EAN-13, UPC-A, or UPC-E.

EAN_TWO_DIGIT_ADD_ON = 'E1'

Two-digit add-on symbol for EAN-13.

EAN_FIVE_DIGIT_ADD_ON = 'E2'

Five-digit add-on symbol for EAN-13.

EAN_13_WITH_ADD_ON = 'E3'

EAN-13, UPC-A, or UPC-E with add-on symbol.

EAN_8 = 'E4'

EAN-8

ITF_14 = 'I1'

ITF-14

GS1_128 = 'C1'

GS1-128

GS1_DATABAR = 'e0'

GS1 DataBar

GS1_COMPOSITE_WITH_SEPARATOR_CHAR = 'e1'

GS1 Composite. Data packet follows an encoded symbol separator character.

GS1_COMPOSITE_WITH_ESCAPE_CHAR = 'e2'

GS1 Composite. Data packet follows an escape mechanism character.

GS1_DATAMATRIX = 'd2'

GS1 DataMatrix

GS1_QR_CODE = 'Q3'

GS1 QR Code

GS1_DOTCODE = 'J1'

GS1 DotCode

classmethod with_ai_element_strings()

Symbologies that may contain AI Element Strings.

Return type
Set[*GS1Symbology*]

classmethod with_gtin()

Symbologies that may contain GTINs.

Return type
Set[*GS1Symbology*]

biip.gs1.**DEFAULT_SEPARATOR_CHARS: Tuple[str] = ('\x1d',)**

The default separator character is <GS>, ASCII value 29.

References

GS1 General Specifications, section 7.8.3.

2.5 biip.gs1.checksums

Checksum algorithms used by GS1 standards.

biip.gs1.checksums.**numeric_check_digit**(*value*)

Get GS1 check digit for numeric string.

Parameters
value (str) – The numeric string to calculate the check digit for.

Return type
int

Returns
The check digit.

Raises
ValueError – If the value isn't numeric.

References

GS1 General Specification, section 7.9

Example

```
>>> from biip.gs1.checksums import numeric_check_digit
>>> numeric_check_digit("950110153100")  # GTIN-13
0
>>> numeric_check_digit("9501234")  # GTIN-8
6
```

biip.gs1.checksums.**price_check_digit**(*value*)

Get GS1 check digit for a price or weight field.

Parameters
value (str) – The numeric string to calculate the check digit for.

Return type
int

Returns
The check digit.

Raises
ValueError – If the value isn't numeric.

References

GS1 General Specification, section 7.9.2-7.9.4

Example

```
>>> from biip.gs1.checksums import price_check_digit
>>> price_check_digit("2875")
9
>>> price_check_digit("14685")
6
```

2.6 biip.gtin

Support for Global Trade Item Number (GTIN).

The *biip.gtin* module contains Biip's support for parsing GTIN formats.

A GTIN is a number that uniquely identifies a trade item.

This class can interpet the following GTIN formats:

- GTIN-8, found in EAN-8 barcodes.
- GTIN-12, found in UPC-A and UPC-E barcodes.
- GTIN-13, found in EAN-13 barcodes.
- GTIN-14, found in ITF-14 barcodes, as well as a data field in GS1 barcodes.

If you only want to parse GTINs, you can import the GTIN parser directly instead of using *biip.parse()*.

```
>>> from biip.gtin import Gtin
```

If parsing succeeds, it returns a *Gtin* object.

```
>>> gtin = Gtin.parse("7032069804988")
>>> gtin
Gtin(value='7032069804988', format=GtinFormat.GTIN_13,
prefix=GS1Prefix(value='703', usage='GS1 Norway'),
company_prefix=GS1CompanyPrefix(value='703206'), payload='703206980498',
check_digit=8, packaging_level=None)
```

A GTIN can be converted to any other GTIN format, as long as the target format is longer.

```
>>> gtin.as_gtin_14()
'07032069804988'
```

As all GTINs can be converted to GTIN-14, it is the recommended format to use when storing or comparing GTINs. For example, when looking up a product associated with a GTIN, the GTIN should first be expanded to a GTIN-14 before using it to query the product catalog.

class biip.gtin.**Gtin**(*value*, *format*, *prefix*, *company_prefix*, *payload*, *check_digit*, *packaging_level=None*)

Data class containing a GTIN.

value: str

Raw unprocessed value.

May include leading zeros.

format: *GtinFormat*

GTIN format, either GTIN-8, GTIN-12, GTIN-13, or GTIN-14.

Classification is done after stripping leading zeros.

prefix: Optional[*GS1Prefix*]

The GS1 Prefix, indicating what GS1 country organization that assigned code range.

company_prefix: Optional[*GS1CompanyPrefix*]

The GS1 Company Prefix, identifying the company that issued the GTIN.

payload: str

The actual payload, including packaging level if any, company prefix, and item reference. Excludes the check digit.

check_digit: int

Check digit used to check if the GTIN as a whole is valid.

packaging_level: Optional[int] = None

Packaging level is the first digit in GTIN-14 codes.

This digit is used for wholesale shipments, e.g. the GTIN-14 product identifier in GS1-128 barcodes, but not in the GTIN-13 barcodes used for retail products.

classmethod parse(*value*, *, *rcn_region=None*, *rcn_verify_variable_measure=True*)

Parse the given value into a *Gtin* object.

Both GTIN-8, GTIN-12, GTIN-13, and GTIN-14 are supported.

Parameters

- **value** (str) – The value to parse.
- **rcn_region** (Union[*RcnRegion*, str, None]) – The geographical region whose rules should be used to interpret Restricted Circulation Numbers (RCN). Needed to extract e.g. variable weight/price from GTIN.
- **rcn_verify_variable_measure** (bool) – Whether to verify that the variable measure in a RCN matches its check digit, if present. Some companies use the variable measure check digit for other purposes, requiring this check to be disabled.

Return type
Gtin

Returns
GTIN data structure with the successfully extracted data. The checksum is guaranteed to be valid if a GTIN object is returned.

Raises
ParseError – If the parsing fails.

as_gtin_8()

Format as a GTIN-8.

Return type
str

as_gtin_12()

Format as a GTIN-12.

Return type
str

as_gtin_13()

Format as a GTIN-13.

Return type
str

as_gtin_14()

Format as a GTIN-14.

Return type
str

without_variable_measure()

Create a new GTIN where the variable measure is zeroed out.

This method is a no-op for proper GTINs. For RCNs, see the method on the *Rcn* subclass.

Return type
Gtin

Returns
A GTIN instance with zeros in the variable measure places.

Raises
EncodeError – If the rules for variable measures in the region are unknown.

class biip.gtin.**GtinFormat**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of GTIN formats.

GTIN_8 = 8

GTIN-8

GTIN_12 = 12

GTIN-12

GTIN_13 = 13

GTIN-13

GTIN_14 = 14

GTIN-14

property length: int

Length of a GTIN of the given format.

class biip.gtin.**Rcn**(*value*, *format*, *prefix*, *company_prefix*, *payload*, *check_digit*, *packaging_level=None*, *usage=None*, *region=None*, *weight=None*, *count=None*, *price=None*, *money=None*)

Restricted Circulation Number (RCN) is a subset of GTIN.

Both RCN-8, RCN-12, and RCN-13 are supported. There is no 14 digit version of RCN.

RCN-12 with prefix 2 and RCN-13 with prefix 02 or 20-29 have the same semantics across a geographic region, defined by the local GS1 Member Organization.

RCN-8 with prefix 0 or 2, RCN-12 with prefix 4, and RCN-13 with prefix 04 or 40-49 have semantics that are only defined within a single company.

Use *biip.gtin.Gtin.parse()* to parse potential RCNs. This subclass is returned if the GS1 Prefix signifies that the value is an RCN.

References

GS1 General Specifications, section 2.1.11-2.1.12

usage: Optional[RcnUsage] = None

Where the RCN can be circulated, in a geographical region or within a company.

region: Optional[RcnRegion] = None

The geographical region whose rules are used to interpret the contents of the RCN.

weight: Optional[Decimal] = None

A variable weight value extracted from the GTIN.

count: Optional[int] = None

A variable count extracted from the GTIN.

price: Optional[Decimal] = None

A variable weight price extracted from the GTIN.

money: Optional['moneyed.Money'] = None

A Money value created from the variable weight price. Only set if py-moneyed is installed and the currency is known.

without_variable_measure()

Create a new RCN where the variable measure is zeroed out.

This provides us with a number which still includes the item reference, but does not vary with weight/price, and can thus be used to lookup the relevant trade item in a database or similar.

This has no effect on RCNs intended for use within a company, as the semantics of those numbers vary from company to company.

Return type
Gtin

Returns
A RCN instance with zeros in the variable measure places.

Raises
EncodeError – If the rules for variable measures in the region are unknown.

class biip.gtin.**RcnUsage**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of RCN usage restrictions.

GEOGRAPHICAL = 'geo'

Usage of RCN restricted to geopgraphical area.

COMPANY = 'company'

Usage of RCN restricted to internally in a company.

class biip.gtin.**RcnRegion**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of geographical regions with custom RCN rules.

The value of the enum is the lowercase ISO 3166-1 Alpha-2 code.

DENMARK = 'dk'

Denmark

ESTONIA = 'ee'

Estonia

FINLAND = 'fi'

Finland

GERMANY = 'de'

Germany

GREAT_BRITAIN = 'gb'

Great Britain

LATVIA = 'lv'

Latvia

LITHUANIA = 'lt'

Lithuania

NORWAY = 'no'

Norway

SWEDEN = 'se'

Sweden

get_currency_code()

Get the ISO-4217 currency code for the region.

Return type

Optional[str]

2.7 biip.sscc

Serial Shipping Container Code (SSCC).

SSCCs are used to identify logistic units, e.g. a pallet shipped between two parties.

If you only want to parse SSCCs, you can import the SSCC parser directly instead of using *biip.parse()*.

```
>>> from biip.sscc import Sscc
```

If parsing succeeds, it returns a *Sscc* object.

```
>>> sscc = Sscc.parse("157035381410375177")
>>> sscc
Sscc(value='157035381410375177', prefix=GS1Prefix(value='570', usage='GS1
Denmark'), company_prefix=GS1CompanyPrefix(value='5703538'),
extension_digit=1, payload='15703538141037517', check_digit=7)
```

Biip can format the SSCC in HRI format for printing on a label.

```
>>> sscc.as_hri()
'1 5703538 141037517 7'
```

If the detected GS1 Company Prefix length is wrong, it can be overridden:

```
>>> sscc.as_hri(company_prefix_length=9)
'1 570353814 1037517 7'
```

class biip.sscc.**Sscc**(*value*, *prefix*, *company_prefix*, *extension_digit*, *payload*, *check_digit*)

Data class containing an SSCC.

value: str

Raw unprocessed value.

prefix: Optional[*GS1Prefix*]

The GS1 Prefix, indicating what GS1 country organization that assigned code range.

company_prefix: Optional[*GS1CompanyPrefix*]

The GS1 Company Prefix, identifying the company that issued the SSCC.

extension_digit: int

Extension digit used to increase the capacity of the serial reference.

payload: str

The actual payload, including extension digit, company prefix, and item reference. Excludes the check digit.

check_digit: int

Check digit used to check if the SSCC as a whole is valid.

classmethod parse(*value*)

Parse the given value into a *Sscc* object.

Parameters
value (str) – The value to parse.

Return type
Sscc

Returns
SSCC data structure with the successfully extracted data. The checksum is guaranteed to be valid if an SSCC object is returned.

Raises
ParseError – If the parsing fails.

as_hri(*, *company_prefix_length=None*)

Render as a human readable interpretation (HRI).

The HRI is often printed directly below barcodes.

The GS1 Company Prefix length will be detected and used to render the Company Prefix and the Serial Reference as two separate groups. If the GS1 Company Prefix length cannot be found, the Company Prefix and the Serial Reference are rendered as a single group.

Parameters
company_prefix_length (Optional[int]) – Override the detected GS1 Company Prefix length. 7-10 characters. If not specified, the GS1 Company Prefix is automatically detected.

Raises
ValueError – If an illegal company prefix length is given.

Return type
str

Returns
A human-readable string where the logic parts are separated by whitespace.

2.8 biip.symbology

Support for Symbology Identifiers.

Symbology Identifiers is a standardized way to identify what type of barcode symbology was used to encode the following data.

The Symbology Identifiers are a few extra characters that may be prefixed to the scanned data by the barcode scanning hardware. The software interpreting the barcode may use the Symbology Identifier to differentiate how to handle the barcode, but must at the very least be able to strip and ignore the extra characters.

Example

```
>>> from biip.symbology import SymbologyIdentifier
>>> SymbologyIdentifier.extract("]E05901234123457")
SymbologyIdentifier(value=']E0', symbology=Symbology.EAN_UPC,
modifiers='0', gs1_symbology=GS1Symbology.EAN_13)
>>> SymbologyIdentifier.extract("]I198765432109213")
SymbologyIdentifier(value=']I1', symbology=Symbology.ITF,
modifiers='1', gs1_symbology=GS1Symbology.ITF_14)
```

References

ISO/IEC 15424:2008.

class biip.symbology.**SymbologyIdentifier**(*value*, *symbology*, *modifiers*, *gs1_symbology=None*)

Data class containing a Symbology Identifier.

value: str

Raw unprocessed value.

symbology: *Symbology*

The recognized symbology.

modifiers: str

Symbology modifiers. Refer to *gs1_symbology* or ISO/IEC 15424 for interpretation.

gs1_symbology: Optional[*GS1Symbology*] = None

If the Symbology Identifier is used in the GS1 system, this field is set to indicate how to interpret the following data.

classmethod extract(*value*)

Extract the Symbology Identifier from the given value.

Parameters
value (str) – The string to extract a Symbology Identifier from.

Return type
SymbologyIdentifier

Returns
Metadata about the extracted Symbology Identifier.

Raises
ParseError – If the parsing fails.

class biip.symbology.**Symbology**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of barcode symbologies that are supported by Symbology Identifers.

References

ISO/IEC 15424:2008, table 1.

CODE_39 = 'A'
Code 39

TELEPEN = 'B'
Telepen

CODE_128 = 'C'
Code 128

CODE_ONE = 'D'
Code One

EAN_UPC = 'E'
EAN/UPC

CODABAR = 'F'
Codabar

CODE_93 = 'G'
Code 93

CODE_11 = 'H'
Code 11

ITF(*Interleaved 2 of 5*) **= 'I'**
ITF (Interleaved 2 of 5)

CODE_16K = 'K'
Code 16K

PDF417 = 'L'
PDF417 and MicroPDF417

MSI = 'M'
MSI

ANKER = 'N'
Anker

CODABLOCK = 'O'

Codablock

PLESSEY_CODE = 'P'

Plessey Code

QR_CODE = 'Q'

QR Code and QR Code 2005

STRAIGHT_2_OF_5_WITH_2_BAR_START_STOP_CODE = 'R'

Straigt 2 of 5 (with two bar start/stop codes)

STRAIGHT_2_OF_5_WITH_3_BAR_START_STOP_CODE = 'S'

Straigt 2 of 5 (with three bar start/stop codes)

CODE_49 = 'T'

Code 49

MAXICODE = 'U'

MaxiCode

OTHER_BARCODE = 'X'

Other barcode

SYSTEM_EXPANSION = 'Y'

System expansion

NON_BARCODE = 'Z'

Non-barcode

CHANNEL_CODE = 'c'

Channel Code

DATA_MATRIX = 'd'

Data Matrix

RSS_EAN_UCC_COMPOSITE = 'e'

RSS and EAN.UCC Composite

OCR(*Optical Character Recognition*) **= 'o'**

OCR (Optical Character Recognition)

POSICODE = 'p'

PosiCode

SUPERCODE = 's'

SuperCode

AZTEC_CODE = 'z'

Aztec Code

2.9 biip.upc

Universal Product Code (UPC).

The *biip.upc* module contains Biip's support for parsing UPC formats.

This class can interpret the following UPC formats:

- UPC-A, 12 digits.
- UPC-E, 6 digits, with implicit number system 0 and no check digit.
- UPC-E, 7 digits, with explicit number system and no check digit.
- UPC-E, 8 digits, with explicit number system and a check digit.

If you only want to parse UPCs, you can import the UPC parser directly instead of using *biip.parse()*

```
>>> from biip.upc import Upc
```

If parsing succeds, it returns a *Upc* object.

```
>>> upc_a = Upc.parse("042100005264")
>>> upc_a
Upc(value='042100005264', format=UpcFormat.UPC_A, number_system_digit=0,
payload='04210000526', check_digit=4)
```

A subset of the UPC-A values can be converted to a shorter UPC-E format by suppressing zeros.

```
>>> upc_a.as_upc_e()
'04252614'
```

All UPC-E values can be expanded to an UPC-A.

```
>>> upc_e = Upc.parse("04252614")
>>> upc_e
Upc(value='04252614', format=UpcFormat.UPC_E, number_system_digit=0,
payload='0425261', check_digit=4)
>>> upc_e.as_upc_a()
'042100005264'
```

UPC is a subset of the later GTIN standard: An UPC-A value is also a valid GTIN-12 value.

```
>>> upc_e.as_gtin_12()
'042100005264'
```

The canonical format for persisting UPCs to e.g. a database is GTIN-14.

```
>>> upc_e.as_gtin_14()
'00042100005264'
```

class biip.upc.**UpcFormat**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of UPC formats.

class biip.upc.**Upc**(*value*, *format*, *number_system_digit*, *payload*, *check_digit=None*)

Data class containing an UPC.

value: str

Raw unprocessed value.

format: *UpcFormat*

UPC format, either UPC-A or UPC-E.

number_system_digit: int

Number system digit.

payload: str

The actual payload, including number system digit, manufacturer code, and product code. Excludes the check digit.

check_digit: Optional[int] = None

Check digit used to check if the UPC-A as a whole is valid.

Set for UPC-A, but not set for UPC-E.

classmethod parse(*value*)

Parse the given value into a *Upc* object.

Parameters
value (str) – The value to parse.

Return type
Upc

Returns
UPC data structure with the successfully extracted data. The checksum is guaranteed to be valid if an UPC object is returned.

Raises
ParseError – If the parsing fails.

as_upc_a()

Format as UPC-A.

Return type
str

Returns
A string with the UPC encoded as UPC-A.

References

GS1 General Specifications, section 5.2.2.4.2

as_upc_e()

Format as UPC-E.

Return type
str

Returns
A string with the UPC encoded as UPC-E, if possible.

Raises
EncodeError – If encoding as UPC-E fails.

References

GS1 General Specifications, section 5.2.2.4.1

as_gtin_12()

Format as GTIN-12.

Return type
str

as_gtin_13()

Format as GTIN-13.

Return type
str

as_gtin_14()

Format as GTIN-14.

Return type
str

CHAPTER

THREE

LICENSE

PYTHON MODULE INDEX

b

INDEX

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W