biip

Stein Magnus Jodal

Sep 14, 2020

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, UPC-A, UPC-E, and ITF-14 barcodes.
- GS1 AI element strings, commonly found in GS1-128 barcodes.

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

CHAPTER

ONE

INSTALLATION

Biip requires Python 3.7 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]"
```

1.1 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.

1.1.1 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("5901234123457")
>>> result
ParseResult(
    value='5901234123457',
    symbology_identifier=None,
    gtin=Gtin(
        value='5901234123457',
        format=GtinFormat.GTIN_13,
        prefix=GS1Prefix(value='590', usage='GS1 Poland'),
        payload='590123412345',
        check_digit=7,
        packaging_level=None,
    ),
    gtin_error=None,
    sscc=None,
```

(continues on next page)

(continued from previous page)

```
    sscc_error="Failed to parse '5901234123457' as SSCC: Expected 18 digits, got 13.",
    gs1_message=None,
    gs1_message_error="Failed to get GS1 Application Identifier from '5901234123457'.
↪",
)
```

1.1.2 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.
- 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.

1.1.3 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)'),
        payload='978149205374',
        check_digit=3,
        packaging_level=None,
    ),
    gtin_error=None,
    sscc=None,
    sscc_error=None,
    gs1_message=None,
```

(continues on next page)

(continued from previous page)

```
    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.

1.1.4 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='123', usage='GS1 US'),
    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)',
    ),
    payload='201112291234',
```

(continues on next page)

(continued from previous page)

```
    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)'
    ),
    payload='201112291234',
    check_digit=6,
    packaging_level=None,
    usage=RcnUsage.GEOGRAPHICAL,
    region=RcnRegion.GREAT_BRITAIN,
    weight=None,
    price=Decimal('12.34'),
    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.

1.1.5 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 `00376130321109103420`:

```
>>> result = biip.parse("00376130321109103420")
>>> result.gs1_message
GS1Message(
    value='00376130321109103420',
    element_strings=[
        GS1ElementString(
            ai=GS1ApplicationIdentifier(
                ai='00',
                description='Serial Shipping Container Code (SSCC)',
                data_title='SSCC',
                fnc1_required=False,
                format='N2+N18',
            ),
            value='376130321109103420',
            pattern_groups=['376130321109103420'],
            gtin=None,
            sscc=Sscc(
                value='376130321109103420',
                prefix=GS1Prefix(value='761', usage='GS1 Schweiz, Suisse, Svizzera'),
                extension_digit=3,
                payload='37613032110910342',
                check_digit=0,
            ),
            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 Schweiz, Suisse, Svizzera'
```

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)376130321109103420'
```

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 `01070320698049881521052610032 9`:

```
>>> result = biip.parse("01070320698049881521052610032 9")
>>> 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)',
            data_title='GTIN',
            fnc1_required=False,
            format='N2+N14',
        ),
        value='07032069804988',
        pattern_groups=['07032069804988'],
        gtin=Gtin(
            value='07032069804988',
            format=GtinFormat.GTIN_13,
            prefix=GS1Prefix(value='703', usage='GS1 Norway'),
            payload='703206980498',
            check_digit=8,
            packaging_level=None,
        ),
        sscc=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'],
        gtin=None,
        sscc=None,
        date=datetime.date(2021, 5, 26),
        decimal=None,
```

(continues on next page)

(continued from previous page)

```
        money=None,
    ),
    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'],
        gtin=None,
        sscc=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.

1.1.6 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!

1.2 API reference

See *Quickstart* for an introduction to how to use the API.

1.2.1 biip

Biip interprets the data in barcodes.

Example

```
>>> import biip
>>> # Ambiguous value that can be interpreted both as a GTIN and a GS1 Message:
>>> result = biip.parse("96385074")
>>> result.gtin
Gtin(value='96385074', format=GtinFormat.GTIN_8,
prefix=GS1Prefix(value='00009', usage='GS1 US'), payload='9638507',
check_digit=4, packaging_level=None)
>>> 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'], gtin=None, sscc=None, date=None, decimal=None,
money=None)])
>>> # Value that is only valid as a GS1 Message:
>>> result = biip.parse("15210526")
>>> result.gtin
None
>>> result.gtin_error
"Invalid GTIN check digit for '15210526': Expected 4, got 6."
>>> result.gs1_message
GS1Message(value='15210526',
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='210526',
pattern_groups=['210526'], gtin=None, sscc=None, date=datetime.date(2021,
5, 26), decimal=None, money=None)])
>>> # Value that cannot be interpreted as any supported format:
>>> 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.
- 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*, *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.

- **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*, *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[*biip.symbology.SymbologyIdentifier*] = None
The Symbology Identifier, if any.

gtin: Optional[biip.gtin._gtin.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.

sscc: Optional[*biip.sscc.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[biip.gs1._messages.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.

1.2.2 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

Example

```
>>> from biip.gs1 import GS1Message
>>> msg = GS1Message.parse("0107032069804988152105261003290")
>>> msg.value
'0107032069804988152105261003290'
>>> msg.as_hri()
'(01)07032069804988(15)210526(10)0329'
>>> len(msg.element_strings)
3
>>> 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'], gtin=Gtin(value='07032069804988',
format=GtinFormat.GTIN_13, prefix=GS1Prefix(value='703', usage='GS1
Norway'), payload='703206980498', check_digit=8, packaging_level=None),
sscc=None, date=None, decimal=None, money=None)
>>> 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'], gtin=None, sscc=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'], gtin=None,
sscc=None, date=None, decimal=None, money=None)
```

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[biip.gs1._element_strings.GS1ElementString]

List of Element Strings found in the message.

classmethod parse(*value*, *, *rcn_region=None*, *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.

- **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.

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*, *gtin=None*, *sscc=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'], gtin=Gtin(value='07032069804988',
format=GtinFormat.GTIN_13, prefix=GS1Prefix(value='703', usage='GS1
Norway'), payload='703206980498', check_digit=8,
packaging_level=None), sscc=None, date=None, decimal=None, money=None)
>>> element_string.as_hri()
'(01)07032069804988'
```

ai: biip.gs1._application_identifiers.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.

gtin: Optional[biip.gtin._gtin.Gtin] = None
A GTIN created from the element string, if the AI represents a GTIN.

sscc: Optional[*biip.sscc.Sscc*] = None
An SSCC created from the element string, if the AI represents a SSCC.

date: Optional[datetime.date] = None
A date created from the element string, if the AI represents a date.

decimal: Optional[decimal.Decimal] = None
A decimal value created from the element string, if the AI represents a number.

money: Optional[moneyed.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*, *separator_chars='\\x1d'*)
Extract the first GS1 Element String from the given value.

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.
- **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("0107032069804988152105261003291")
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 GS1Prefix

Returns Metadata about the extracted prefix.

Raises ***ParseError*** – If the parsing fails.

class biip.gs1.**GS1Symbology**(*value*)

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]

1.2.3 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
```

1.2.4 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.

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

Example

```
>>> from biip.gtin import Gtin
>>> gtin = Gtin.parse("5901234123457")
>>> gtin
Gtin(value='5901234123457', format=GtinFormat.GTIN_13,
prefix=GS1Prefix(value='590', usage='GS1 Poland'),
payload='590123412345', check_digit=7, packaging_level=None)
>>> gtin.as_gtin_14()
'05901234123457'
```

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

Data class containing a GTIN.

value: str

Raw unprocessed value.

May include leading zeros.

format: biip.gtin._enums.GtinFormat

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

Classification is done after stripping leading zeros.

prefix: biip.gs1._prefixes.GS1Prefix

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

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*)

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** (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.

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

class biip.gtin.**GtinFormat**(*value*)
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
Length of a GTIN of the given format.

Return type int

class biip.gtin.**Rcn**(*value*, *format*, *prefix*, *payload*, *check_digit*, *packaging_level=None*)
Restricted Circulation Number (RCN) is a subset of GTIN.

RCNs with prefix 02 and 20-29 have the same semantics across a geographic region, defined by the local GS1 Member Organization.

RCNs with prefix 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.

usage: Optional[biip.gtin._enums.RcnUsage] = None
Where the RCN can be circulated, in a geographical region or within a company.

region: Optional[biip.gtin._enums.RcnRegion] = None
The geographical region whose rules are used to interpret the contents of the RCN.

weight: Optional[decimal.Decimal] = None
A variable weight value extracted from the barcode, if indicated by prefix.

price: Optional[decimal.Decimal] = None
A variable weight price extracted from the barcode, if indicated by prefix.

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 new 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*)
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*)
Enum of geographical regions with custom RCN rules.

BALTICS = 'baltics'
Baltics (Estonia, Latvia, Lithuania)

GREAT_BRITAIN = 'gb'
Great Britain

NORWAY = 'no'
Norway

SWEDEN = 'se'
Sweden

get_currency_code()
Get the ISO-4217 currency code for the region.

Return type Optional[str]

1.2.5 biip.sscc

Serial Shipping Container Code (SSCC).

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

Example

```
>>> from biip.sscc import Sscc
>>> sscc = Sscc.parse("376130321109103420")
>>> sscc
Sscc(value='376130321109103420', prefix=GS1Prefix(value='761',
usage='GS1 Schweiz, Suisse, Svizzera'), extension_digit=3,
payload='37613032110910342', check_digit=0)
>>> sscc.as_hri()
'3 761 3032110910342 0'
>>> sscc.as_hri(company_prefix_length=8)
'3 761 30321 10910342 0'
```

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

Data class containing an SSCC.

value: str

Raw unprocessed value.

prefix: biip.gs1._prefixes.GS1Prefix

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

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.

Parameters **company_prefix_length** (Optional[int]) – Length of the assigned GS1 Company prefix. 7-10 characters. If not specified, the GS1 Company Prefix and the Serial Reference are rendered as a single group.

Raises **ValueError** – If an illegal company prefix length is used.

Return type str

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

1.2.6 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: *biip.symbology.Symbology*
The recognized symbology.

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

gs1_symbology: Optional[biip.gs1._symbology.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.o

class biip.symbology.**Symbology**(*value*)
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 = '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 = 'o'
OCR (Optical Character Recognition)

POSICODE = 'p'
PosiCode

SUPERCODE = 's'
SuperCode

AZTEC_CODE = 'z'
Aztec Code

1.3 Changes

1.3.1 v0.5.2 (2020-09-04)

- Bugfix: Add zero in front of GTIN-12 before extracting GS1 Prefix. GTIN-12s start with U.P.C. Company Prefixes, which have to be padded with a zero in front to convert them to valid GS1 Company Prefixes.
- Change the error message in the top level parser's *ParseError* to include the name of data type we failed to parse.

1.3.2 v0.5.1 (2020-09-03)

- Bugfix: *biip.gtin.Rcn.without_variable_measure()* returns the instance unchanged for RCNs intended for usage within a company. Previously, this crashed as *region* was unset for company RCNs.

1.3.3 v0.5.0 (2020-09-03)

- **Breaking change:** Change argument *separator_char* to *separator_chars* in plural, accepting an iterable of multiple characters.

1.3.4 v0.4.0 (2020-08-31)

- Add *Quickstart* guide.

biip

- **Breaking change:** Change return value of *parse()* from `Union[Gtin, GS1Message]` to *ParseResult*.
- If a Symbology Identifier is present, select parser based on it.
- Improved parsing error messages.

biip.sscc

- Add *Sscc* class to parse, validate, and format Serial Shipping Container Codes (SSCC).

biip.symbology

- Add support for parsing and stripping Symbology Identifiers.

1.3.5 v0.3.1 (2020-08-21)

biip

- Strip surrounding whitespace before selecting parser.

1.3.6 v0.3.0 (2020-08-21)

biip.gs1

- Add *filter()* to find all parsed Element Strings that match the criteria.
- Add *get()* to find first parsed Element String that matches the criteria.
- Add *decimal* field which is set for AIs with weight, volume, dimensions, dicount percentages, and amounts payable.
- Add *money* field which is set for AIs with both amounts payable and currency. This field is only set if the optional dependency `py-moneyed` is installed.
- Strip surrounding whitespace before parsing.

biip.gtin

- Detect Restricted Circulation Numbers (RCN) and return a subclass of *Gtin*, *Rcn*, with additional fields and helpers for working with RCNs.
- Classify an RCN as being restricted to either a geographical region or to a company.
- Support interpreting RCNs according to varying rules depending on the geographical region specified by the user.
- Support for zeroing out the variable measure part, to help with looking up trade items in a database or similar.
- Add RCN rules for the Baltics, Great Britain, Norway, and Sweden.
- Strip surrounding whitespace before parsing.
- Bugfix: Keep all leading zeros in GTIN-8.
- Bugfix: Convert GTIN-8 to GTIN-12 before extracting GS1 Prefix.

1.3.7 v0.2.1 (2020-08-19)

biip.gtin

- Raise *ParseError* if there is less than 8 or more than 14 significant digits in the barcode.

1.3.8 v0.2.0 (2020-08-19)

biip

- *parse()* can parse GTIN and GS1-128 data.

biip.gs1

- *GS1Message* can parse GS1-128 data.
- *GS1ApplicationIdentifier* recognizes all 480 existing GS1 AIs.
- *GS1Prefix* recognizes all existing GS1 prefixes.
- *checksums* has functions to calculate check digits for numeric data and price/weight fields.

biip.gtin

- Support for validating, parsing, and converting between GTIN-8, GTIN-12, GTIN-13, and GTIN-14.

1.3.9 v0.1.0 (2020-05-20)

Initial release to reserve the name on PyPI.

CHAPTER

TWO

LICENSE

CHAPTER

THREE

PROJECT RESOURCES

- Documentation
- Source code
- Issue tracker

PYTHON MODULE INDEX

b

INDEX

G

I

L

M

N

O

P

Q

R

S

T

U

V

W