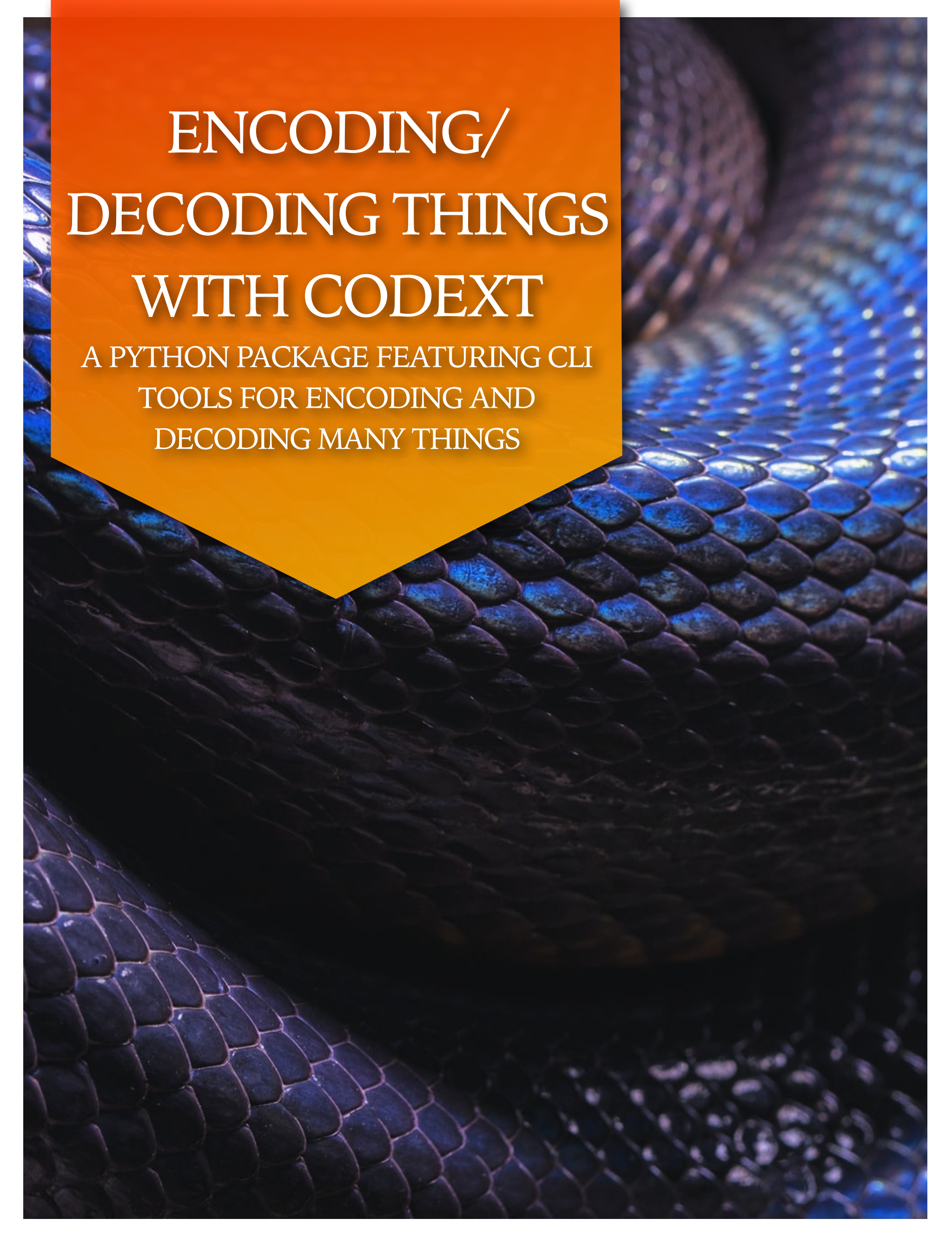
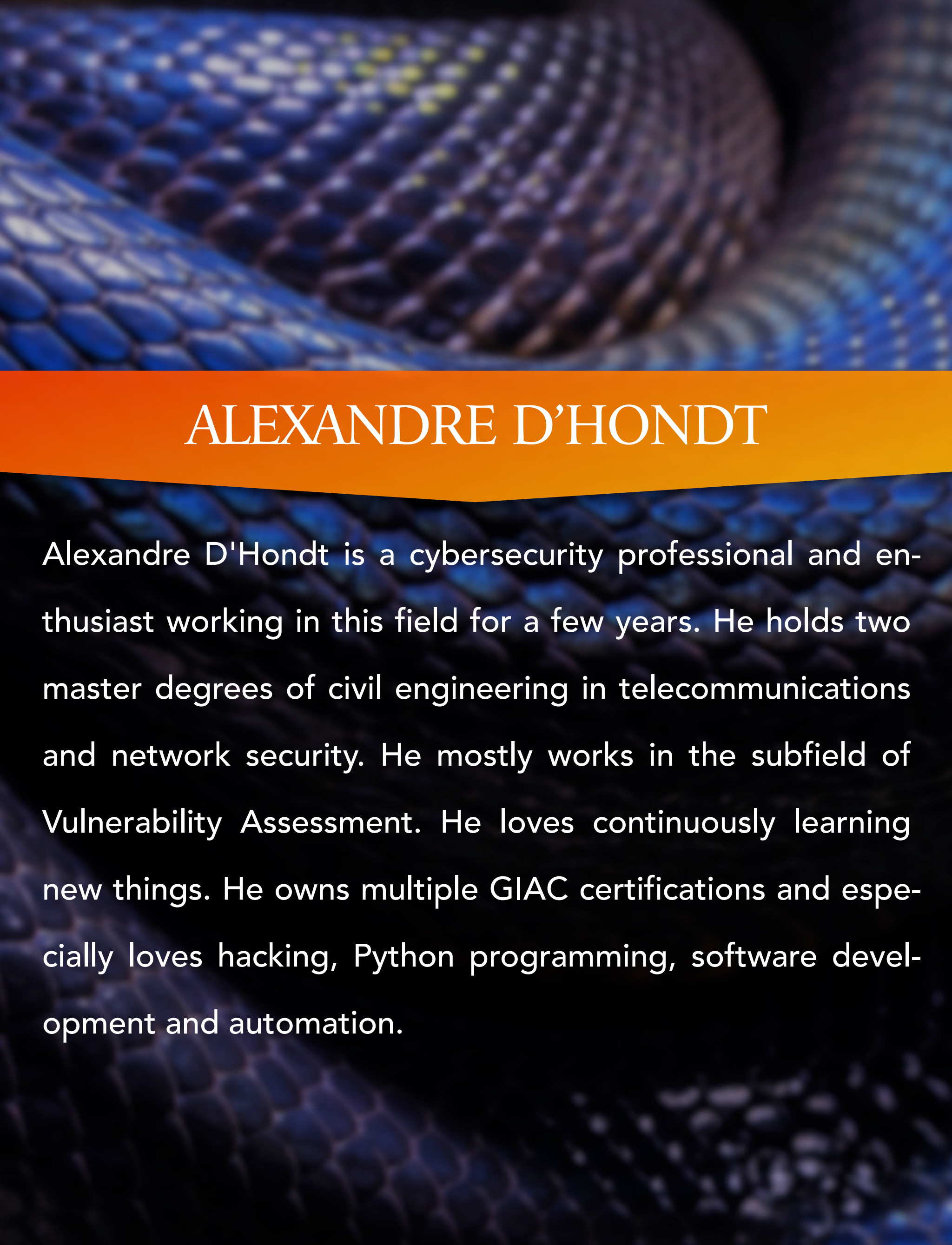




ENCODING/ DECODING THINGS WITH CODEXT

A PYTHON PACKAGE FEATURING CLI
TOOLS FOR ENCODING AND
DECODING MANY THINGS



ALEXANDRE D'HONDT

Alexandre D'Hondt is a cybersecurity professional and enthusiast working in this field for a few years. He holds two master degrees of civil engineering in telecommunications and network security. He mostly works in the subfield of Vulnerability Assessment. He loves continuously learning new things. He owns multiple GIAC certifications and especially loves hacking, Python programming, software development and automation.

Abstract

Python provides a native package for handling encodings called `codecs`. It has a neat API defining codecs for encoding/decoding with various well-known encodings, especially for dealing with special characters. However, it contains a limited set of codecs and does not handle multi-layered encoded inputs.

That is where `codext` comes into play, the *CODecs EXTension*. It provides various features for easily enriching the registry of codecs from the native library, increases this with many new encodings, and provides multi-layer guessing relying on an artificial intelligence algorithm.

This article explains its basics and presents some of its capabilities.

1. Introduction

Encoding or decoding data is a common operation, especially when dealing with special characters like Cyrillic or Chinese. These kinds of encodings are handled by a native library in Python called `codecs`. In security, we can also encrypt something and then base64-encode it to transform the ciphertext to a limited set of printable characters. This is something also handled by `codecs`. However, its registry of codecs is relatively limited and mostly contains classical encodings for special characters.

Therefore, providing a programming interface for manipulating the registry of codecs is a first challenge. Afterwards, it could be interesting to provide a guess feature for addressing multi-layer encodings. This can be handled by an artificial intelligence algorithm like a tree search, optimized with a scoring heuristic for ranking best-matching encodings. All these challenges are addressed in `codext` [1], the *CODecs EXTension*.

The remainder of this article presents the enhanced API and the extended registry of codecs. Then it explains the guess feature and the tuning of its parameters for refining a search, converging towards the right decoded output. Finally, it shows the related Command-Line Interface tools shipped with the package with some usage examples.

2. Codecs registry

The **registry** of codecs is the structure that contains the definitions of codecs. First, this subsection presents how to inspect existing codecs and then **how codecs can be added or removed** from this registry.

2.1. Inspecting codecs

When looking at the source code of `codecs`, we can see that it imports various objects from `_codecs` (a shared object), including a `lookup` function for returning a `CodecInfo` instance holding the attributes of a codec and its encode/decode functions. This lookup works by walking the registry, an ordered list of so-called *search functions*. These functions, when called with a string, return matching `CodecInfo` instances. This way, while walking this registry, the first *search function*

returning an instance (when the search does not match, nothing is returned) terminates the lookup, regardless if another function also matches further in the list. The interesting point here is that this design allows defining flexible patterns for codecs. For instance, we can define the Base64 codecs as matched by *base64*, *base-64* or *base_64*. This is certainly more handy for the end-user but this mostly allows for something even more interesting; **parametrization** (see in the next subsection).

Some first improvements to `codecs` as defined with `codext` include the following:

- It defines its own **registry**, relying on this one first before the native registry. This allows it to supersede existing codecs.
- The enhanced **lookup** function returns a `CodecInfo` instance with more attributes, namely defining a category for each codec, based on its origin folder (*native* for original codecs, or subfolder names from the `codext` package for the new codecs).

```
>>> codext.lookup("base45")
<codecs.CodecInfo object for encoding base45 at 0x7f0910696a00>
```

Figure 1: Example of using the "lookup" function

- A **list** function allows a list of all the existing codecs. Specifying categories as arguments allows filtering out codecs. **list_categories** shows the list of available categories.

```
>>> codext.list()
['affine', 'ascii', 'ascii85', 'atbash', 'bacon', ..., 'base36', 'base58', 'base62', , ...]
```

```
>>> codext.list("binary")
['baudot', 'baudot-spaced', 'baudot-tape', 'bcd', 'bcd-extended0', 'bcd-extended1', ...]
```

Figure 2: Example of using the "list" function

- A **search** function allows filtering out codecs based on a regular expression, considering the patterns of each *search function*. This means that, for instance, `search("5$")` will match every encoding name ending with "5", including dynamic ones, like *rot* (*rot-5*), *XOR* (*xor-5*), etc.

```
>>> codext.search("baudot")
['baudot', 'baudot_spaced', 'baudot_tape']
>>> codext.search("a1")
['capitalize', 'octal', 'octal_spaced', 'ordinal', 'ordinal_spaced', 'radio']
```

Figure 3: Example of using the "search" function

2.2. Manipulating codecs

From there, we can list, lookup or search for codecs. Nevertheless, we need to be able to add or remove new encodings if we want to enrich the registry. From the source code of `codexts`, we find a `register` function that allows registering new *search functions*. In order to facilitate adding new codecs, `codext` defines a new `add` function that will take the essential parameters as arguments and set additional attributes to the returned `CodecInfo` instances, attributes that will be necessary for more advanced features like guessing.

Some improvements or codec manipulation features of `codext` include the following:

- The `add` function takes, as its most important arguments, the encoding name, encode/decode functions and the pattern it can match to. The example hereafter shows an example of its usage, showing the definition of the encode and decode functions to be associated to a new codec named *mycodec*. Note that this is a very simple example that does not handle a pattern but we could, for instance, define “`^my[_] codec$`”. Such a pattern allows for **flexible encoding names**.

```
import codext

def mycodec_encode(text, errors="strict"):
    # do some encoding stuff
    return encoded, len(text)

def mycodec_decode(text, errors="strict"):
    # do some decoding stuff
    return decoded, len(text)

codext.add("mycodec", mycodec_encode, mycodec_decode)
```

Figure 4: Example of adding a new codec with "add"

But, as mentioned before, using a pattern also allows for **parametrization**. Let us take a weak cryptographic algorithm like Barbie Typewriter [2], holding only four possible character sets selected by a number from 1 to 4, in other words a kind of 2-bits key; this is equivalent to the following encodings : *barbie-1*, ..., *barbie-4*. Therefore, the encoding name for this codec requires a parameter taking values from 1 to 4. This is achieved with a regular expression “`^barbie[_]?([1-4])$`”. This way, the codec named *barbie* is matched by a string containing its name and a number from 1 to 4.

```
import codext

def mydyncodec_encode(i):
    def encode(text, error="strict"):
        # do something depending on i
        return result, len(text)
    return encode

codext.add("mydyncodec", mydyncodec_encode, pattern=r"mydyn-(\d+)$")
```

Figure 5: Example of adding a parametrized codec with “add”

In the example above, we see that the pattern defines a digit parameter. Note that, while working, this way of defining the parameter is too broad, the best solution being to restrict it to only “`[12]`”.

The complete guide of how to write encode/decode functions and add a new codec can be found in the documentation of `codext` [3].

- While a generic `add` function allows defining a codec with any kind of user-defined encode/decode function, it can be very repetitive in the code when dealing with encoding maps. The `add_map` function then allows entering map parameters, including a dictionary either as the only map or as pattern-map items or even a list of maps (as shown in the next figure).

```
import codext

ENCMAP = [
    {'00': "A", '01': "B", '10': "C", '11': "D"},
    {'00': "D", '01': "C", '10': "B", '11': "A"},
]

codext.add("mydyncodec", ENCMAP, "#", ignore_case=True, intype="bin", pattern=r"mydyn-([12])$")
```

Figure 6: Example of adding a new encoding map with "add_map"

In the example above, we see a digit parameter like in the previous example. Beware that the parameter starts at 1 while the index is set from 0 for selecting the encoding map in the `ENCMAP` list. It is also possible to define a replacement character when using the encode/decode function with an `errors` argument set to "replace" (the default value being "strict", meaning that when an error occurs, the encoding/decoding stops with an error). The codec can also be set to ignore case while encoding or decoding. The input and output types can be specified (amongst the set of built-in type conversion functions, i.e. `bin`, `bool`, `str`, `ord`) through the `intype` and `outype` arguments.

The complete guide of how to write encoding maps and add a new map codec can be found in the documentation of `codext` [3].

- Multiple additional functions holding self-explanatory names allow removing or restoring codecs, including the `clear` function for clearing the `codext`'s internal registry (not this of codecs), the `remove` function for removing specific codecs matching the input regular expression and the `reset` function that restores the whole codecs to the initial state.

```
>>> codext.clear()
>>> codext.encode("test", "bin")

Traceback (most recent call last):
[...]
LookupError: unknown encoding: bin
```

```
>>> codext.reset()
>>> codext.encode("test", "bin")
'01110100011001010111001101110100'
```

Figure 7: Example of using the removal and restore functions

- Finally, the `examples` function allows generating lists of examples of valid encoding names from the given regular expression. This is particularly useful with parametrized codecs that are more complex to take in hand.

```
>>> codext.examples("rot")
['rot-14', 'rot-24', 'rot-7', 'rot18', 'rot3', 'rot4', 'rot6', 'rot_1', 'rot_12', 'rot_2']
>>> codext.examples("dna")
['dna-1', 'dna-2', 'dna-5', 'dna1', 'dna4', 'dna5', 'dna6', 'dna8', 'dna_3', 'dna_5']
>>> codext.examples("barbie", 5)
['barbie-1', 'barbie1', 'barbie4', 'barbie_2', 'barbie_4']
```

Figure 8: Example of generating lists of example encoding names

These new functions make the API of `codext`, enhanced regarding this of `codecs`, and that is bound on the `codecs` module too. Note that using `codecs.register` will also register the new *search function* in `codecs`, while `codext.register` will not.

2.3. New encodings

Beyond offering an enhanced API for manipulating codecs, `codext` provides a myriad of new encodings. While extending the registry since January 2020 when the project was born, the quantity of codecs has grown so far that it was better to categorize them for easier filtering (handled in the `list` function) and also for applying selections while using the guess feature (see in the next section). Categories are defined through the folders (right-stripped from their “s”) inside the `codext` package, the native codecs being put in a special category logically called *native*. Another special category called *custom* holds the codecs added manually at runtime.

Up to now, `codext` contains the following categories (other than *native* and *custom*):

- ✓ **Base:** This handles all kinds of bases, defining every possible generic one but also specifying particular bases found in the wild. Therefore, it includes various classical and trickier bases such as 2 (binary), 3, 4, 8, 16 (hexadecimal), 26, 36, 45 [4], 58 (bitcoin, ripple and flickr), 62, 63, 64, 85, 91 [5], 100 [6] and 122 [7].
- ✓ **Binary:** This contains binary-related codecs like the Baudot code, Binary Coded Decimal (BCD), Excess-3 code (aka Stibitz code), the Gray code, Manchester and rotate-N-bits.
- ✓ **Common:** This category undertakes common codecs like a1z26 (just mapping letters to their index in the alphabet) but also octal and ordinal, for the sake of porting built-in conversion functions (`ord`, `oct`) to codecs in a way usable with others. It also includes “dummy” codecs performing simple string operations like reverse, uppercase, etc.
- ✓ **Compression:** Another interesting category includes some compression algorithms like GZIP and PKZIP, featuring LZMA, Bzip2 and Deflate but also LZ77 [8] and LZ78 [9].
- ✓ **Crypto:** This category features many cryptography algorithms turned into parametrized codecs, including Affine, Atbash and Bacon ciphers, Barbie TypeWriter [2], CTX1 Password encoding, Scytale and the very common ciphers Rot-N (including Caesar), Shift-byte-ordinal and XOR-N (handling a key with a single character).
- ✓ **Language:** This category includes some popular languages like Braille, Morse code and Tom-Tom but also lorem ipsum, leetspeak, Navajo code talker, the radio phonetic alphabet or the language of the South Park character Kenny.

- ☑ *Other*: This regroups uncategorized codecs, including DNA sequences, HTML entities, consonants and vowels indices, Markdown and URL-encode.
- ☑ *Stegano*: This category is still in its early stage and currently contains the Polybius-based Klopff code, resistor colors, SMS (Nokia 3310) and whitespace-based codecs.

These new encodings provide a lot of examples that eventual contributors can rely on.

3. Guess feature

Decoding inputs without knowing the right codec a priori is something new with `codext`. This feature, aimed to guess the encoding used, relies on a **ranking of the potential codecs** according to characteristics of the given input. This also **handles multi-layers encoded inputs** by leveraging a simple artificial intelligence algorithm called the **tree search**. This subsection explains how it is implemented and how to use it for converging towards the right decoded output.

3.1. Candidate codecs ranking

This part of `codext`'s internals is still a work in progress but the current heuristic takes multiple string characteristics into account for scoring and ranking the potential codecs used for encoding.

These characteristics include:

- The *entropy*: Using the Shanon entropy of the input allows giving a bonus to codecs that present a similar entropy. Given that the input is long enough, this can help put the best candidate codecs higher in the final ranking. However, if the entropy is not matching, it does not imply a penalty. When the entropy is close to the target one of the candidate codec, a linear penalty is applied as of the deviation to the reference entropy, with a threshold to 0 for not leading to a penalty. Another special case is when the candidate codec does not modify the input entropy while encoding. In this case, the `entropy` argument when using the `add` function for this codec can be set to a lambda identity function. Using such a function also lends itself to setting an entropy relationship between the input and output, e.g. using a linear formula.
- The *length of the character set*: When the input has a character set whose length exactly matches that of the candidate set, it is better scored, therefore putting it higher in the ranking.
- Presence of a *padding* character: Some codecs use padding (e.g. Base2^N) while others may have the same character in their character set. In this case, a characteristic discriminates by giving a bonus to the candidate codec that indeed uses a padding character while such a character is present at the end of the input.
- A *penalty*: Some candidate codecs have other characteristics that often match even when unrelated, therefore causing junk candidates in the final ranking. These codecs, once empirically evaluated, may be set to get a penalty in scoring when declaring them with the `add` function.

This scoring heuristic, certainly far from perfect, is very empirical and is a first attempt to get something generic. It performs well in some cases, such as the base encodings. However, either the characteristics per codec could be refined by evaluating their behavior with a large set of test patterns, or a machine-learning model may replace the entire heuristic in the near future after further dedicated research.

3.2. Tree search

The aforementioned heuristic allows guessing a single layer of encoding. What if we want to guess **multiple layers of encodings** from a given input? For this problem, we need an artificial intelligence algorithm for performing an exhaustive search of every possible combination up to a maximum depth we define, choosing to stop the algorithm when a predefined criterion is met. The typical algorithm for this purpose is the tree search.

In `codext`, this search for the multiple layers of encoding is achieved by using the **breadth-first tree search** algorithm. It stops when a given condition is met in the form of a function (called the *stop function*) applied to the decoded string at the current depth. The `guess` function returns the *decoded string* and a tuple with the related *encoding names* in order of application.

The following parameters can be entered for tuning the search:

- The *stop function* (`stop_func`): This can be a function or a regular expression to be matched (automatically converted to a function that uses the `re` module when the input string for this parameter does not match a pre-built stop function); by default, checks if all input characters are printable.

Some useful functions are embedded in a dedicated submodule called `stopfunc`, importable from the scope of `codext`;

- `stopfunc.printables`: This checks if all characters are printable.
- `stopfunc.text`: This checks for printable characters but also for a maximum entropy threshold defined on an empirical basis.
- `stopfunc.flag`: This uses a regular expression pattern considering declinations of the word “*flag*”, including variants like *fl@g*, *f14g*, *Flag*, etc.
- `stopfunc.lang_XX`: This relies on the package `langdetect` [10] (if installed separately), accepting a myriad of possible language bigrams (instead of `XX`).
- The *minimum* and *maximum* search *depths* (`min_depth`, `max_depth`): These determine the minimum or maximum depth for the tree search, by default respectively 0 or 5.
- The specific *categories of codecs* to be selected (`codec_categories`): This can be a string indicating a codec category or a list of category strings; by default, `None`, meaning the whole categories are considered.

- The list of *codecs to be applied* before starting the search (`found`): This is set as a list or tuple of currently found encodings. This can be used to save time if the first decoding steps are already known; by default, an empty tuple.

3.3. Search refinement

In many cases, while the scoring heuristic may rank potential codecs with a decent effectiveness, it will fail to find the right output at a first glance most of the time. This is inherent to the great variety of encodings supported by `codext`. Therefore, we need to make maximum use of prior information for refining the search.

For instance, knowing that an input has been encoded with only base codecs, we can restrict the search to the *base* category by setting the `codec_categories` parameter. This will accelerate the search and directly provide the right decoded output, as shown in the example below. Note that the first example shows “*Barbie*” as the codec as the decoded output satisfies the stop condition from the default stop function, which checks for printable characters.

```
>>> codext.guess("CJG3Ix8bVcSRML0qwDUg28aDsT7")
('FKU2Ng7lJbR>.IHuzLDv17eLhE6', ('barbie',))
```

```
>>> codext.guess("CJG3Ix8bVcSRML0qwDUg28aDsT7", codec_categories="base")
('This is a test', ('base62', 'base64'))
```

Figure 9: Example of setting the `codec_categories` parameter

If we know even more prior information, e.g. we expect an output in English text, we can refine the search using a *stop function*, as shown in the following example. Note that, as mentioned in the previous subsection, using a stop function based on a language will only work if `langdetect` [10] was installed separately (the reason for this choice is that `codext` is aimed to install as few dependencies as possible).

```
>>> codext.guess("CJG3Ix8bVcSRML0qwDUg28aDsT7", codext.stopfunc.lang_en, codec_categories="base")
('This is a test', ('base62', 'base64'))
```

Figure 10: Example of using a stop function

If we know even more, e.g. we expect a stopword like “*a*” or “*the*” in the output, we can set it between whitespaces. If we expect a word, like “*test*” as shown in the following example, we can specify it instead of the stop function and it will be used as a regular expression.

```
>>> codext.guess("CJG3Ix8bVcSRML0qwDUg28aDsT7", "test", codec_categories="base")
('This is a test', ('base62', 'base64'))
```

Figure 11: Example of using a string instead of a stop function

In some cases, it could also be useful to specify minimum and/or maximum search depths. Let us assume that the first stage of decoding will produce, given the default stop function (printables), a valid string and we expect multiple stages, we can set `min_depth` to something higher. In the same way, given the default value of 5 for `max_depth`, if we expect more stages, we must increase this parameter for getting the right result.

4. CLI tools

In order to ease its use, `codext` gets installed with a few **command-line tools**. This subsection briefly shows them with some examples of their use.

4.1. Main tool

The main tool, called like the package, holds commands bound to the following aforementioned API functions: `encode`, `decode`, `guess`, `rank` and `search`. For the sake of user-friendliness, encodings can be chained, as shown in the following example. The standard input or a file (with the `--input-file` option) can be used as input and the output is displayed in the terminal or can be saved to a file (with the `--output-file`).

```
$ echo "test string" | codext encode base26
EUCTMOZTHTEJTTOQXBR
$ echo "test string" | codext encode base26 caesar xor-5
CSAPKUDPLPCNPPUW\FV
$ echo "CSAPKUDPLPCNPPUW\FV" | codext decode xor-5 caesar
EUCTMOZTHTEJTTOQXBR
$ echo "CSAPKUDPLPCNPPUW\FV" | codext decode xor-5 caesar base26
test string
```

Figure 12: Examples of encoding/decoding with the `codext` tool

The next example shows the use of the `rank` command for guessing the next encoding layer and then the use of the `guess` command, taking advantage of the rank 1 encoding of the previous step and the prior knowledge of the word “*test*” in the expected decoded output.

```
$ echo "4DGFhKkf0dcTJk3LRSL556KDNUMrKl" | codext rank
[+] 0.98904: base62
[+] 0.98620: base63
[+] 0.97502: base67
[+] 0.68801: whitespace+after-before
[+] 0.68801: whitespace-after-before
[+] 0.63869: base85
[+] 0.62601: citrix-ctx1
[+] 0.60899: base91
[+] 0.60899: base91-inv
[+] 0.60899: base91-alt
$ echo "4DGFhKkf0dcTJk3LRSL556KDNUMrKl" | codext guess base62 -f test
Codecs: base62, rot-1, manchester
test string
```

Figure 13: Example of ranking and guess-decoding with the `codext` tool

4.2. Base and debase

The package also features a series of base tools that mimic and replace the Linux ones, `base32` and `base64`, including 2, 3, 4, 8, 16, 32, 32-z, 32-hex, 32-geohash, 36, 45, 58-bitcoin, 58-ripple, 58-flickr, 62, 63, 64, 64-url 67, 85, 91, 100, 122.

It also provides a guessing tool called `debase`, which relies on the `guess` feature but restricted to the *base* category. The following example shows some usage examples of these tools.

```
$ echo "test string" | base58-bitcoin | base122 | base85 | base45
7S8TK4GG618CN1C4Y80C8Y69FL4 7D667BH6

$ echo "7S8TK4GG618CN1C4Y80C8Y69FL4 7D667BH6" | debase
D;#E2V`U_0EFA|Ha#dh48s2~

$ echo "7S8TK4GG618CN1C4Y80C8Y69FL4 7D667BH6" | debase -f lang_en
test string
```

Figure 14: Examples of encoding/decoding with the `baseX` and `debase` tools

Conclusion

In this article, we presented CodExt [1], a Python library extending the native `codecs` package with many new features. This extension reshapes and enhances the original API and provides a myriad of new encodings. It also adds multi-layer codec guessing by relying on a common artificial intelligence algorithm called the breadth-first tree search, backed by a scoring heuristic leveraging some codec characteristics for ranking best-matching codec candidates. Finally, it extends the native solution with some handy console scripts.

In the near future, we plan to evaluate the characteristics of the scoring heuristics on all the implemented codecs in order to optimize their parameters and increase the performance. Machine learning algorithms could also be addressed in the search for a more efficient heuristic.

References:

1. *Python codecs extension*, A. D'Hondt, GitHub, available at <https://github.com/dhondta/python-codext>, last visited October 2021
2. *Barbie™ Typewriter*, Cryptomuseum, available at <https://www.cryptomuseum.com/crypto/mehano/barbie>, last visited October 2021
3. *Codext – Extension of native codecs for Python*, A. D'Hondt, ReadTheDoc, available at <https://python-codext.readthedocs.io/en/latest>, last visited October 2021
4. *The Base45 Data Encoding*, P. Falstrom, IETF, available at <https://datatracker.ietf.org/doc/html/draft-falstrom-base45-04.txt>, last visited October 2021

5. *base91 encoding*, Joachim Henke, SourceForge, available at <http://base91.sourceforge.net>, last visited October 2021
6. *Base100*, Aleksandr, GitHub, available at <https://github.com/MasterGroosha/pybase100>, last visited October 2021
7. *Base122*, K. Albs, GitHub, available at <https://github.com/kevinAlbs/Base122/blob/master/base122.js>, last visited October 2021
8. *A Python LZ77-Compressor*, W. Manassra, GitHub, available at <https://github.com/manassra/LZ77-Compressor>, last visited October 2021
9. *Lempel Ziv Compression*, M. Watson, GitHub, available at <https://github.com/mileswatson/lempel-ziv-compression>, last visited October 2021
10. *Langdetect – Language detection library ported from Google's language-detection*, PyPi, available at <https://pypi.org/project/langdetect>, last visited October 2021