```
>>> "naïve" == "naïve"
True
```

```
>>> "naïve" == "naïve"
True
>>> "naïve" == "naïve"
False
```

```
>>> "naïve" == "naïve"
True
>>> "naïve" == "naïve"
False
>>> "A" == "A"
False
```

```
>>> for c in "HELLO":
...     print(c)


O
L
L
E
H
```

```
>>> "Piñata"[:3]
'Pin'
>>> "Piñata"[3:]
'̃ata'
```

```
>>> int("8")
4
```

Why `len("🤯") == 4` and other weird things you should know about strings in Python

Marie Roald & Yngve Mardal Moe

Slides:
github.com/MarieRoald/PyConUS25

Encoding strings

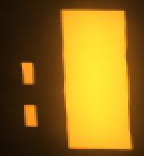
Comparing strings

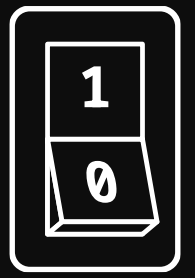
Slicing strings

Slides:
github.com/MarieRoald/PyConUS25

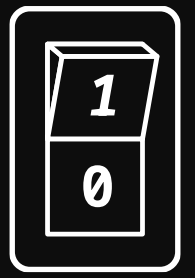
PART I

Encoding strings

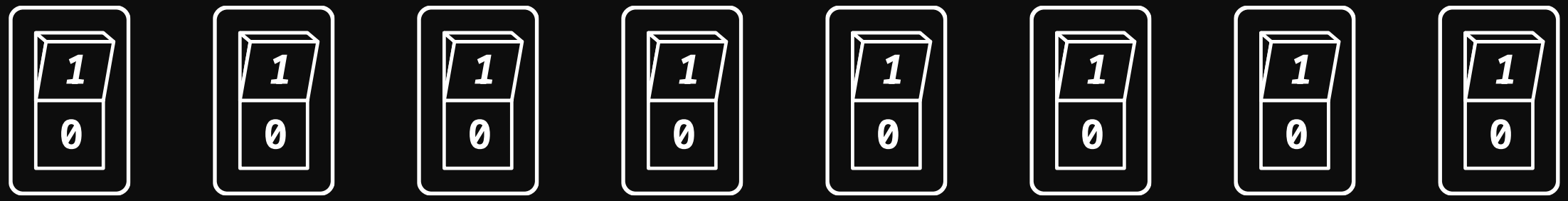




1

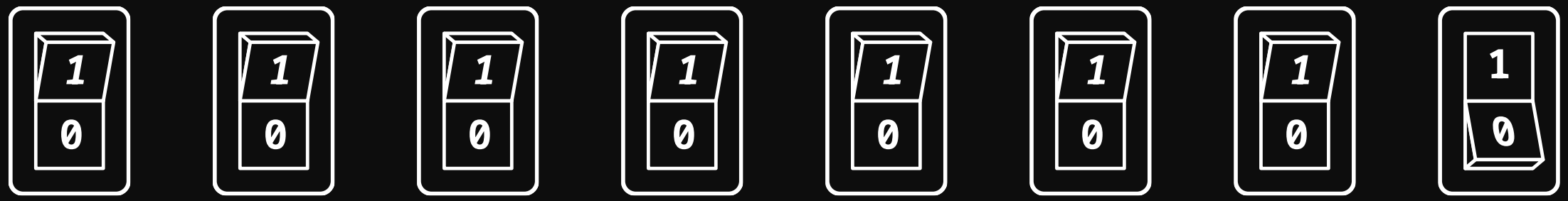


0



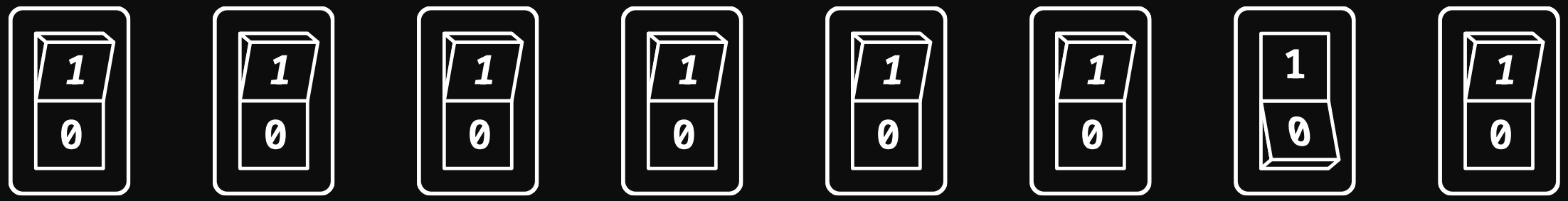
0 0 0 0 0 0 0 0

0



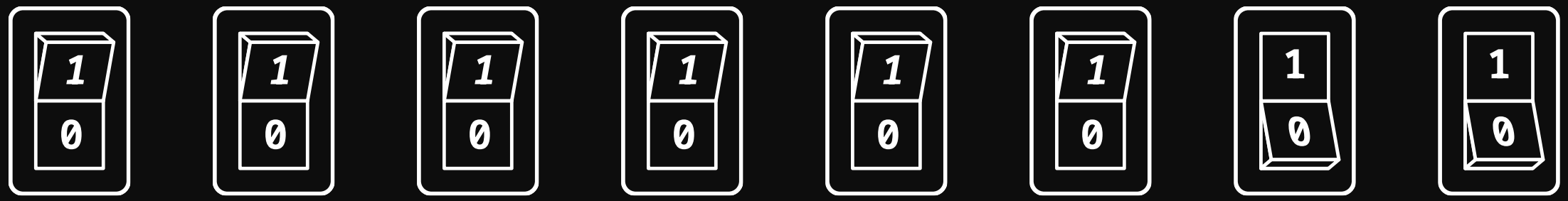
0 0 0 0 0 0 0 1

1



0 0 0 0 0 0 1 0

2



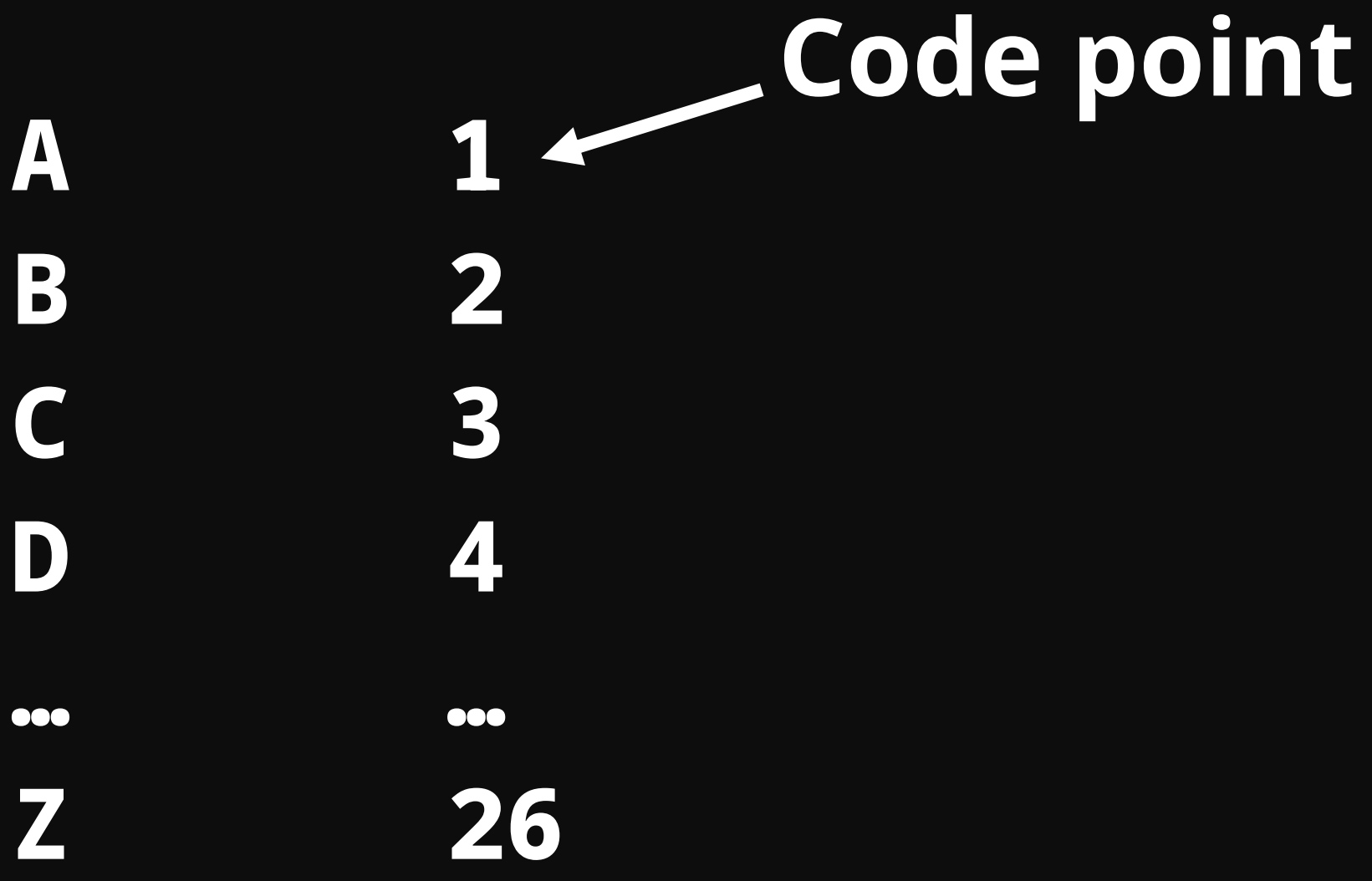
0 0 0 0 0 0 1 1

3

By enumerating letters, we can store them as integers

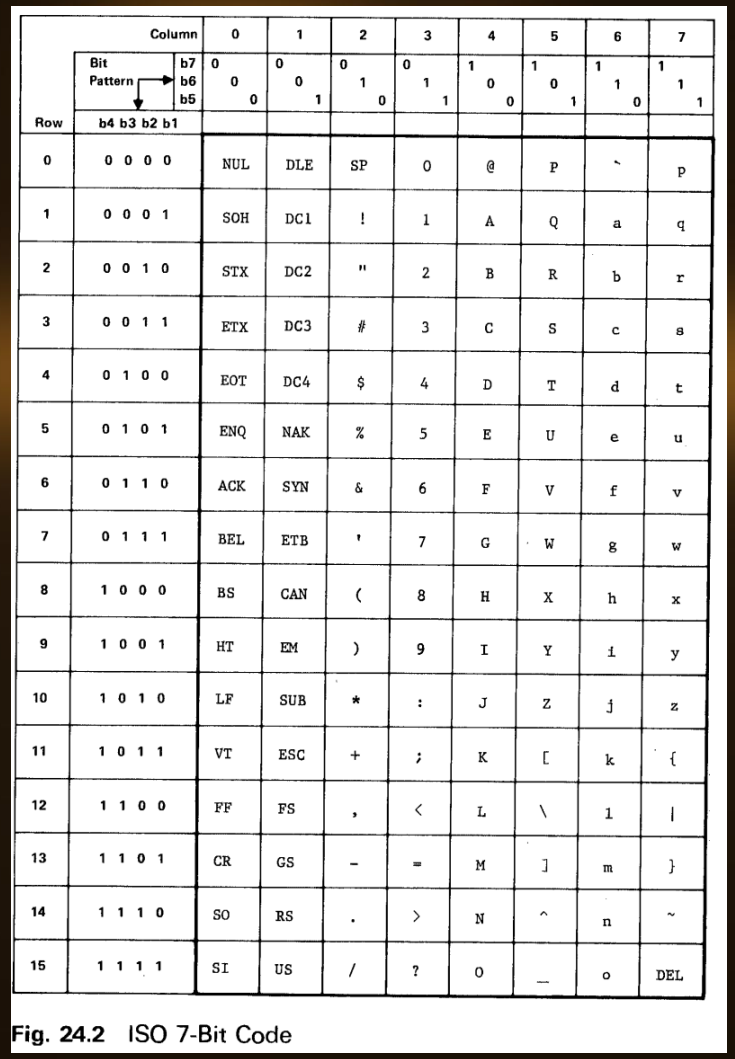
A	1
B	2
C	3
D	4
…	…
Z	26

By enumerating letters, we can store them as integers




Dette er en
morsom vits
~
~
(END)
ørttw rw rn
mÅwsÅm vrts
~
~
(END)

The ASCII standard from 1963 from contains 128 code points



Column		0	1	2	3	4	5	6	7
Bit Pattern → b7 b6 b5		0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Row	b4 b3 b2 b1								
0	0 0 0 0	NUL	DLE	SP	0	@	P	`	p
1	0 0 0 1	SOH	DC1	!	1	A	Q	a	q
2	0 0 1 0	STX	DC2	"	2	B	R	b	r
3	0 0 1 1	ETX	DC3	#	3	C	S	c	s
4	0 1 0 0	EOT	DC4	$	4	D	T	d	t
5	0 1 0 1	ENQ	NAK	%	5	E	U	e	u
6	0 1 1 0	ACK	SYN	&	6	F	V	f	v
7	0 1 1 1	BEL	ETB	'	7	G	W	g	w
8	1 0 0 0	BS	CAN	(	8	H	X	h	x
9	1 0 0 1	HT	EM	)	9	I	Y	i	y
10	1 0 1 0	LF	SUB	*	:	J	Z	j	z
11	1 0 1 1	VT	ESC	+	;	K	[	k	{
12	1 1 0 0	FF	FS	,	<	L	\	l	|
13	1 1 0 1	CR	GS	-	=	M	]	m	}
14	1 1 1 0	SO	RS	.	>	N	^	n	~
15	1 1 1 1	SI	US	/	?	O	_	o	DEL

Fig. 24.2 ISO 7-Bit Code

Charles E. MacKenzie, *Coded Character Sets: History and Development*. (1979)


Ostehøvel

KONGERIKET NORGE

PATENT NR. 43377

STYRET FOR DET INDUSTRIELLE RETSVERN

Gjör vitterlig: *Med hjemmel av lov om patenter av 2 den juli 1910 med tilleggslov av 9 juli 1923 har Styret git*

Snekkermester Thor Björklund
av Lillehammer

patent paa Kniv til skjæring av ost

overensstemmende med den vedheftede fremstilling.

Patenttiden løper fra 27 februar 1925,

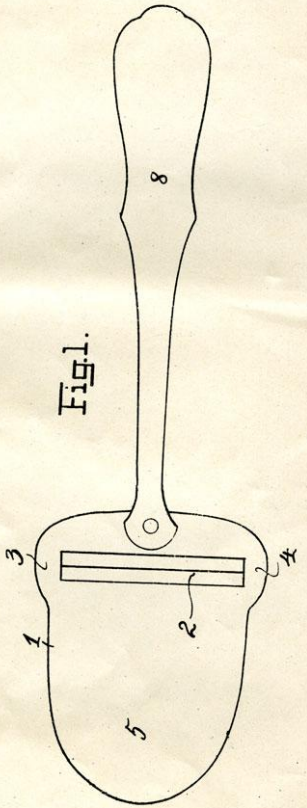
da krav om patent indkom til Styret for det Industrielle Retsvern.

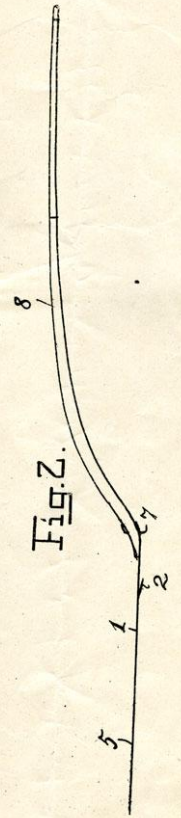
Oslo den 15 november 1926

STYRET FOR DET INDUSTRIELLE RETSVERN OSLO

K 14508

Patent nr. 43377





```
>>> "ostehøvel".encode("ascii")
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec
can't encode character '\xf8' in
position 5: ordinal not in range(128)
```

Since 1963, we have seen many language-specific standards for encoding text

Standard Encodings

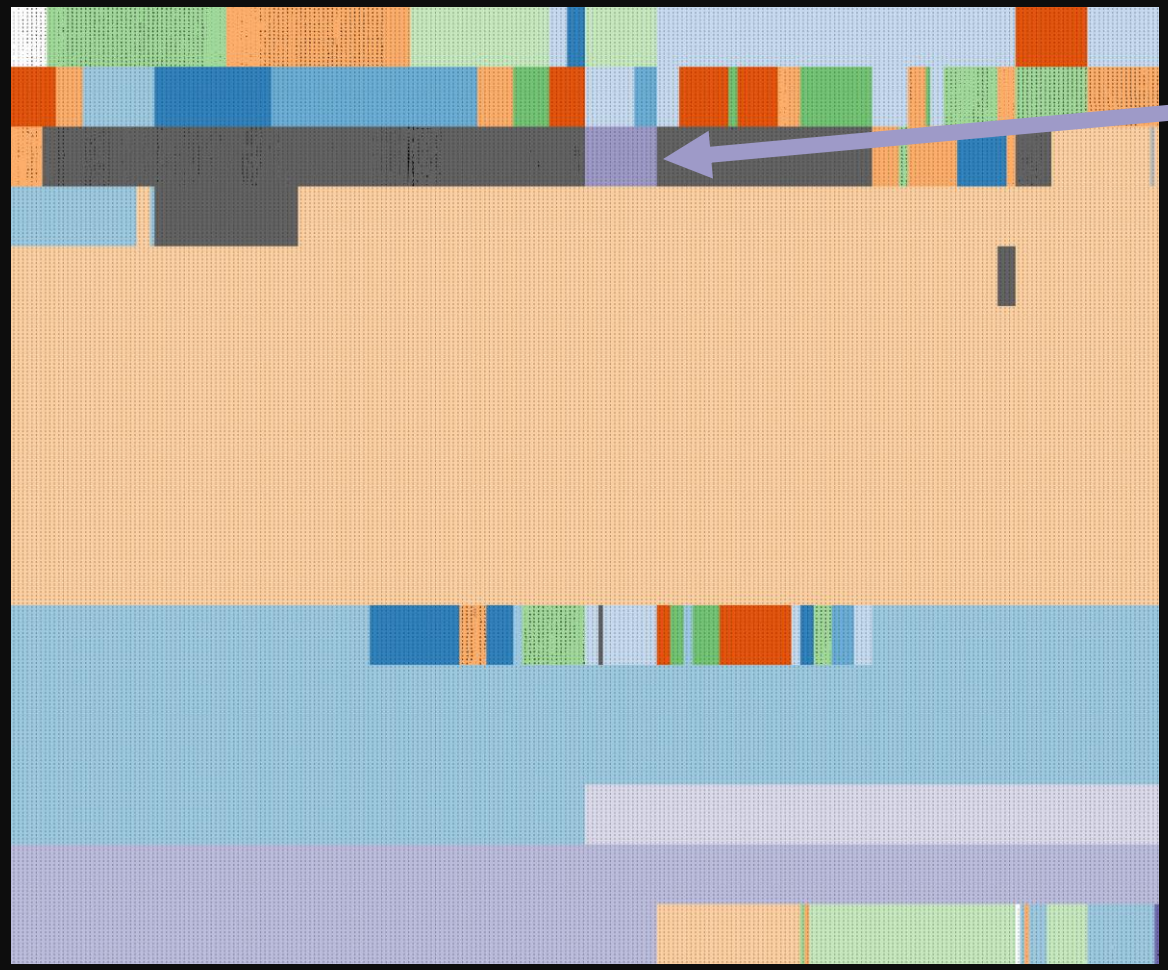
Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German *Added in version 3.4.*
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek

[...]

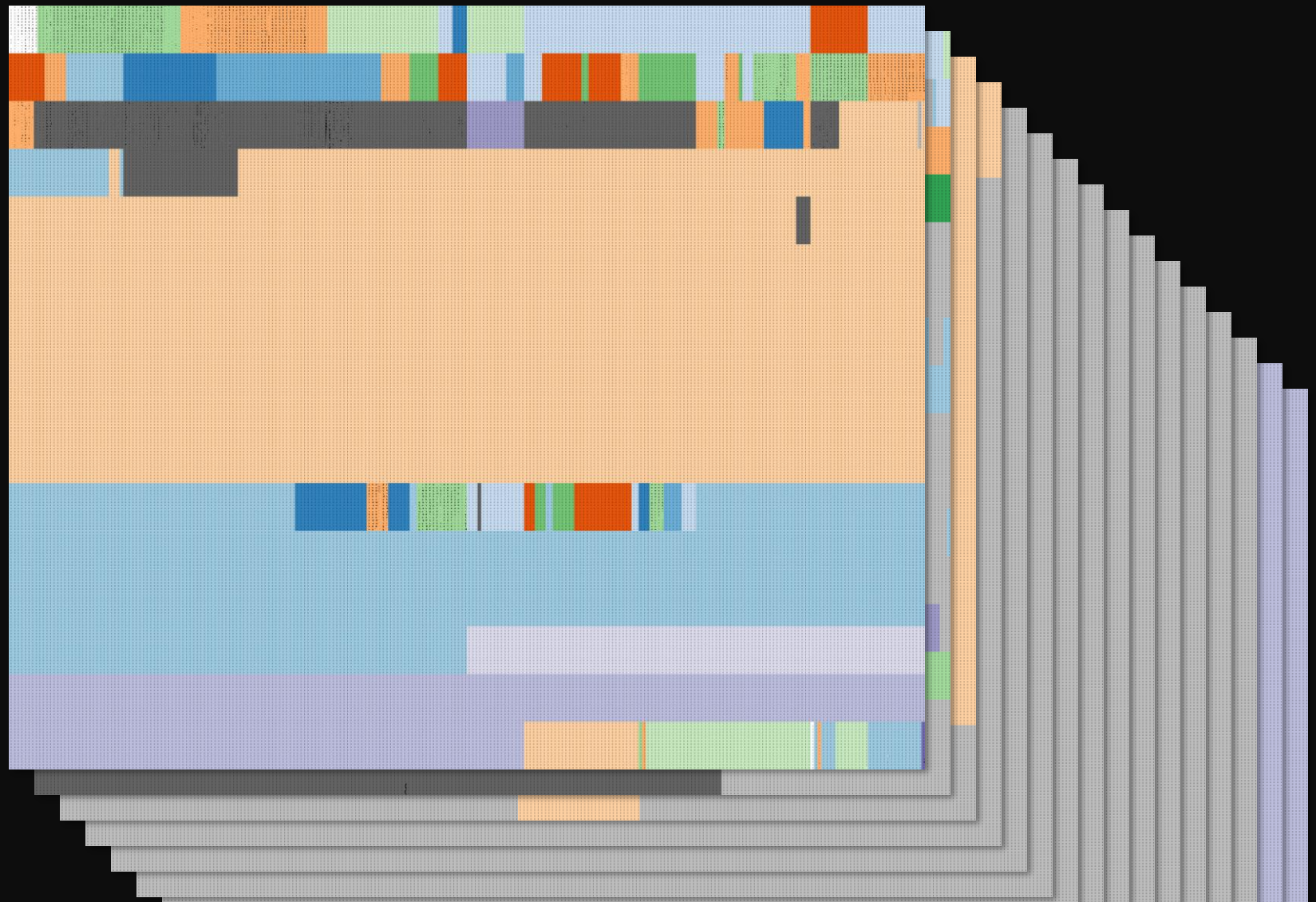
87 text encoding schemes

https://docs.python.org/3.13/library/codecs.html#standard-encodings

By using two bytes per code point, we can store 65 536 different characters



By using four bytes per code-point Unicode supports more than one million characters



UTF-8 strings take approximately four times more space than ASCII strings

```
>>> len("pycon".encode("ascii"))
5


>>> len("pycon".encode("utf-32"))
24
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
0000000 =    0
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
0000001 =   1
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
00000010 =   2
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
01111111 = 127
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
11000010 10000000 = 128
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
11000010 10000001 = 129
```

By using the first bit(s) of each code-point, we can encode characters compactly

```
11000010 10000010 = 130
```

The UTF-8 encoding can store ASCII-strings compactly

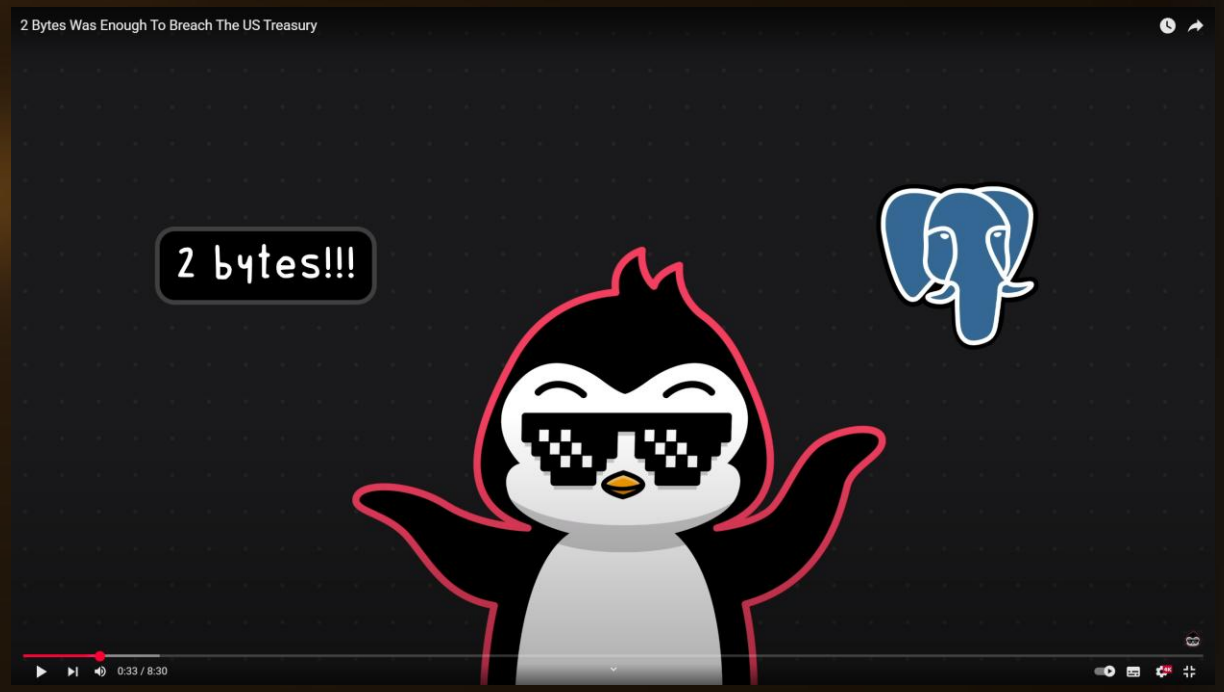
```
>>> len("pycon".encode("ascii"))
5


>>> len("pycon".encode("utf-32"))
24


>>> len("pycon".encode("utf-8"))
5
```

The variable-length encoding makes UTF-8 much more difficult to parse correctly

The Register®

Fewer said: "Because of how PostgreSQL string escaping routines handle invalid UTF-8 characters, in combination with how invalid byte sequences within the invalid UTF-8 characters are processed by psql, an attacker can leverage CVE-2025-1094 to generate a SQL injection."

Running meta-commands can extend psql's functionality, and it's through these that an attacker can feasibly achieve ACE by using the exclamation mark meta-command to execute a shell command on the operating system. Attackers can also use the vulnerability to execute SQL statements of their choosing.



https://www.theregister.com/2025/02/14/postgresql_bug_treasury/

https://www.youtube.com/watch?v=rgsIkZkflMw (PwnFunction)

Python has three types of unicode string encodings

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

```
type Py_UCS4
type Py_UCS2
type Py_UCS1
```

Part of the Stable ABI.

These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use Py_UCS4.

> *Added in version 3.3.*

https://docs.python.org/3/c-api/unicode.html

If all characters in a string are in ASCII, then each code point is stored with one byte

```
>>> sys.getsizeof("a"*4097+"\u0010")
1042
```

If just one code point needs two bytes, then all code points are stored using two bytes

```
>>> sys.getsizeof("a"*4097+"\u0010")
1042
>>> sys.getsizeof("a"*4097+"\u0100")
2060
```

If just one code point needs more than two bytes, then all are stored using four bytes

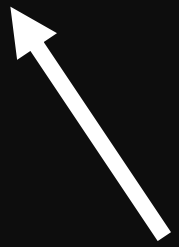
```
>>> sys.getsizeof("a"*4097+"\u0010")
1042
>>> sys.getsizeof("a"*4097+"\u0100")
2060
>>> sys.getsizeof("a"*4097+"🐢")
4064
```

Each Unicode character is part of a category

```
>>> from unicodedata import category
>>> category("8")
'Nd'
```

Each Unicode character is part of a category

```
>>> from unicodedata import category
>>> category("8")
'Nd'
```



Decimal number

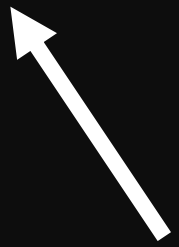
```
>>> int("8")
4
```

We can get more information about characters by looking at their names

```
>>> from unicodedata import name
>>> name("৪")
'BENGALI DIGIT FOUR'
```

Unicode code-points can be used to do more than just encode characters

```
>>> from unicodedata import category
>>> category("HELLO"[0])
'Cf'
```



Format

Unicode code-points can be used to do more than just encode characters

```
>>> from unicodedata import name
>>> name("HELLO"[0])
'LEFT-TO-RIGHT EMBEDDING'
```

Unicode code-points can be used to do more than just encode characters

```
>>> from unicodedata import name
>>> name("HELLO"[0])
'LEFT-TO-RIGHT EMBEDDING'
>>> name("HELLO"[1])
'RIGHT-TO-LEFT OVERRIDE'
```

Text-formatting characters can sometimes lead to surprising results

```
>>> print("HELLO"[2:])
OLLEH
```

Part II

Comparing strings

Do you approach the dragon? [y/n]

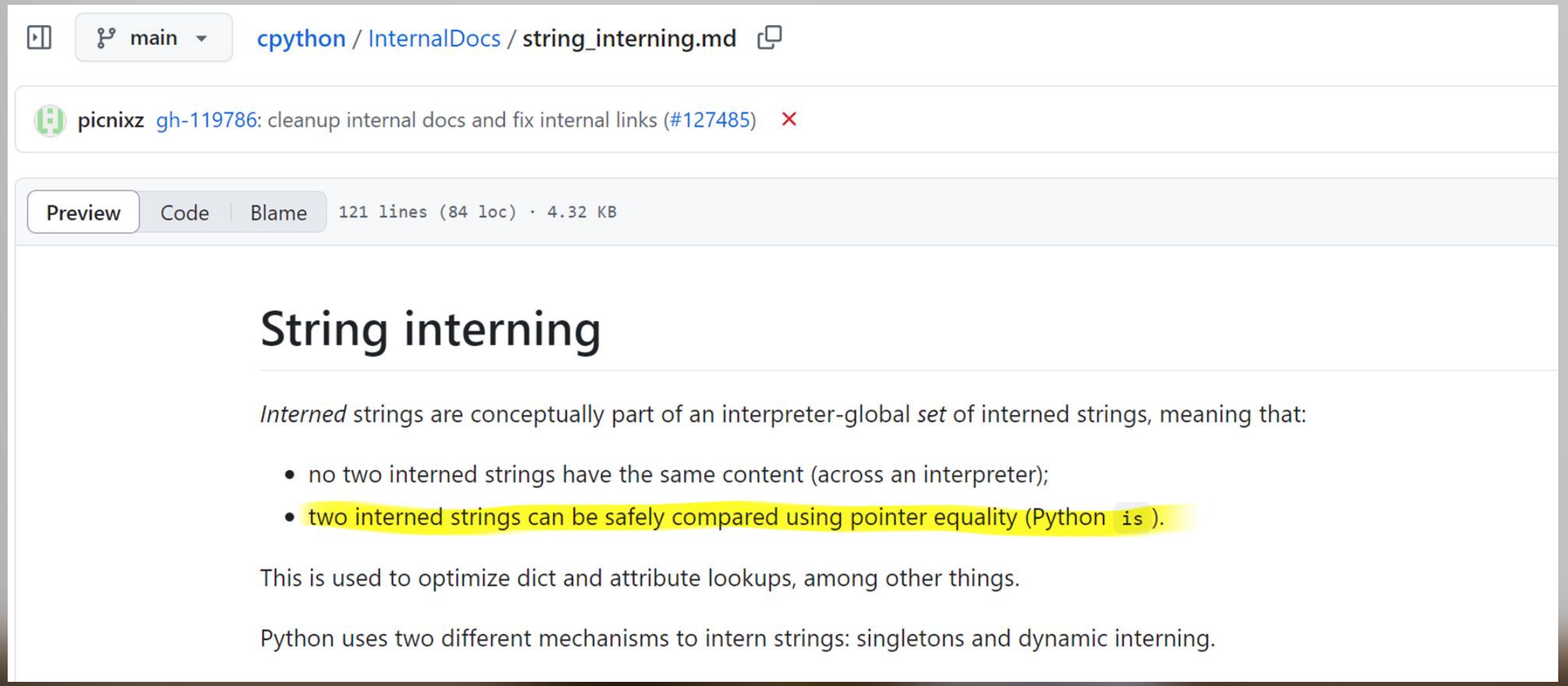
```
>>> answer = input()
>>> answer is "y"
```

```
>>> "y" is "y"
```

```
>>> "y" is "y"
01_is_compare.py:1: SyntaxWarning:
"is" with 'str' literal. Did you
mean "=="?
```

```
>>> "y" is "y"
01_is_compare.py:1: SyntaxWarning:
"is" with 'str' literal. Did you
mean "=="?
True
```

To save time and memory, Python stores some strings just once in memory



main cpython / InternalDocs / string_interning.md

picnixz gh-119786: cleanup internal docs and fix internal links (#127485)

Preview Code Blame 121 lines (84 loc) · 4.32 KB

String interning

Interned strings are conceptually part of an interpreter-global *set* of interned strings, meaning that:

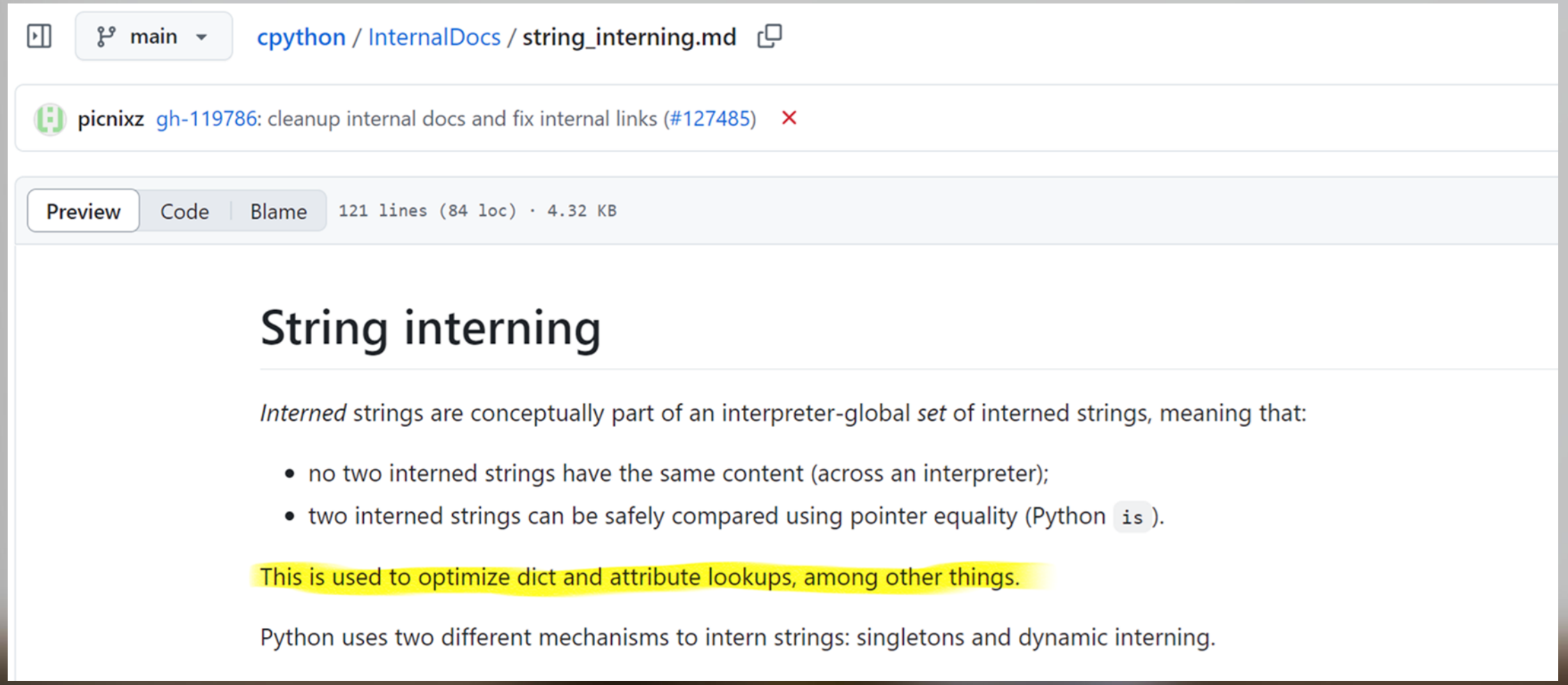
- no two interned strings have the same content (across an interpreter);
- two interned strings can be safely compared using pointer equality (Python `is`).

This is used to optimize dict and attribute lookups, among other things.

Python uses two different mechanisms to intern strings: singletons and dynamic interning.

https://github.com/python/cpython/blob/main/InternalDocs/string_interning.md

To save time and memory, Python stores some strings just once in memory



main cpython / InternalDocs / string_interning.md

picnixz gh-119786: cleanup internal docs and fix internal links (#127485)

Preview Code Blame 121 lines (84 loc) · 4.32 KB

String interning

Interned strings are conceptually part of an interpreter-global *set* of interned strings, meaning that:

- no two interned strings have the same content (across an interpreter);
- two interned strings can be safely compared using pointer equality (Python `is`).

This is used to optimize dict and attribute lookups, among other things.

Python uses two different mechanisms to intern strings: singletons and dynamic interning.

https://github.com/python/cpython/blob/main/InternalDocs/string_interning.md

Not all strings are interned by Python

```
>>> "a"*4097 is "a"*4097
False
```

The equal operator checks if the contents of two strings are equal

```
>>> "y" == "y"
True
```

Sometimes, we want to compare strings without caring about case

```
>>> "Y" == "y"
False
```

Sometimes, we want to compare strings without caring about case

```
>>> "Y".lower() == "y"
True
```

+43471320300 | weissensee@ktn.gde.at | Parteienverkehr: (in dieser Ze...

A+

Gemeinde Weißensee

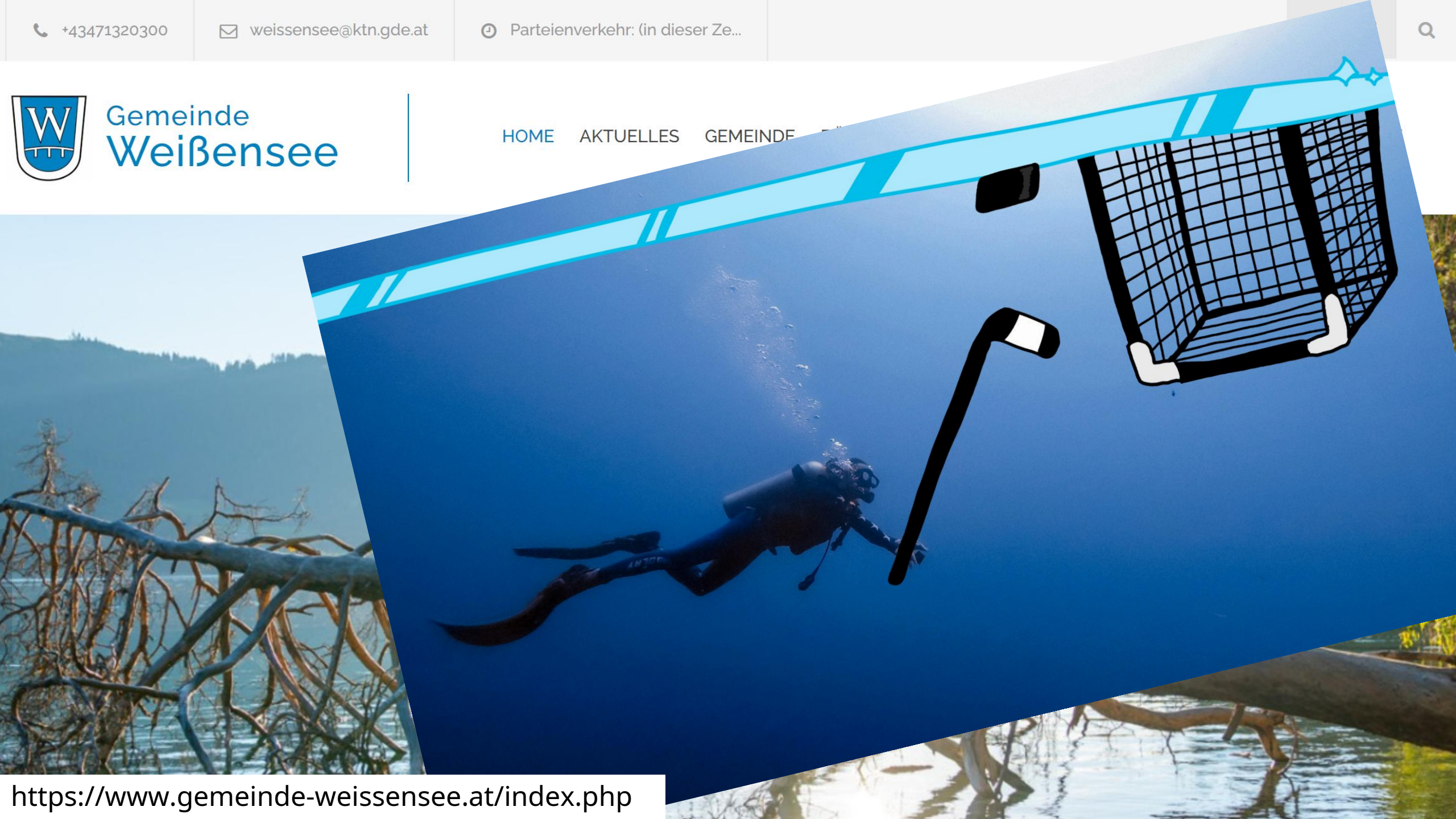
HOME AKTUELLES GEMEINDE BÜRGERSERVICE STANDESAMT POLITIK ÜBER'N WEISSENSEE

https://www.gemeinde-weissensee.at/index.php

+43471320300 | weissensee@ktn.gde.at | Parteienverkehr: (in dieser Ze...

Gemeinde Weißensee

HOME AKTUELLES GEMEINDE

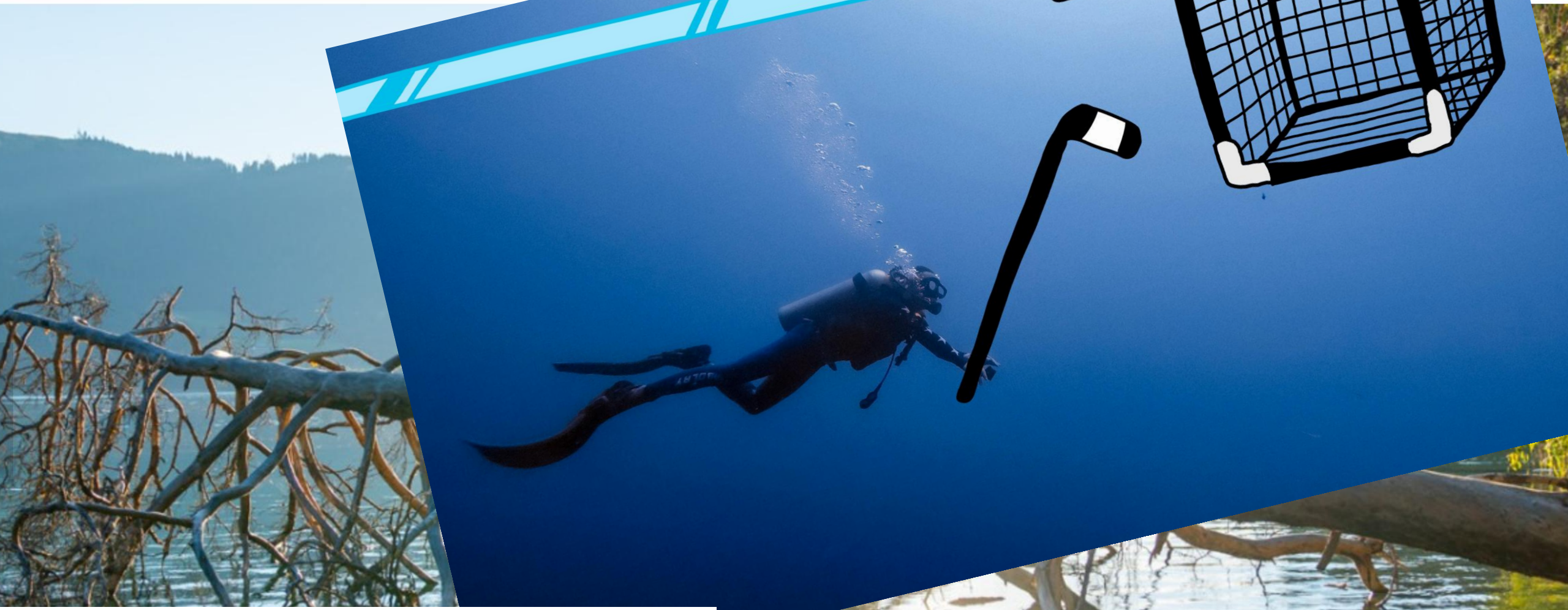


https://www.gemeinde-weissensee.at/index.php

+43471320300 weissensee@ktn.gde.at Parteienverkehr: (in dieser Ze...

Gemeinde Weißensee

HOME AKTUELLES GEMEINDE



https://www.gemeinde-weissensee.at/index.php

Using case transformations to compare strings is not robust

```
>>> "Weissensee".upper() ==
    "Weißensee".upper()
True
```

Using case transformations to compare strings is not robust

```
>>> "Weissensee".lower() ==
    "Weißensee".lower()
False
```

We should casefold for caseless string comparison

3.13.3 Default Case Folding

Case folding is related to case conversion. However, the main purpose of case folding is to contribute to caseless matching of strings, whereas the main purpose of case conversion is to put strings into a particular cased form.

Default Case Folding does not preserve normalization forms. A string in a particular Unicode normalization form may not be in that normalization form after it has been casefolded.

The Unicode Standard, Version 16.0 – Core specification.

Python has builtin support for casefolding

```
>>> "Weissensee".casefold() ==
    "Weißensee".casefold()
True
```

D R T ⁂ O i S ⍜ Ⱡ У A J E ᯈ Ꝑ Ⱥ Ⱶ Г Ꙩ W
ძ Ր G M ᖇ ᯅ Ⲟ H Ȝ Ꙗ Ɵ ʯ G Ո h Z ᖁ Ꝏ I ᔕ
ᯒ ᯛ ᯡ Ɛ Ⱨ ᘐ 4 Ь ᡮ ᡏ R Ꮮ Ꮃ Ꮥ Ꮾ Ꭻ Ꭻ V S ᛕ
ஃ Ꮯ L C ᯎ ᯒ P Ꮯ Ᏼ Ꮒ K ძ Ꮯ Ꮆ ᭐ᭀ ᭄ ᭐ 6 ᔕ
ᵬ ᶈ ᖵ Ᏻ B Ꮆ ß ᾽Aι᾽Aι᾽Aι᾽Aι᾽Aι᾽Aι᾽Aι᾽Aι᾽Hι᾽Hι᾽Hι᾽Hι᾽Hι
᾽Hι᾽Hι᾽Hι᾽Ωι᾽Ωι᾽Ωι᾽Ωι᾽Ωι᾽Ωι᾽Ωι᾽Ωι Aι Hι Ωιμ ß ᾽n ſ ǰ ̣
ΐ ΰ ς β θ φ π ϰ ρ ε ⅼ ϐ ϗ ϧ Ᏻ в ᏼ ϭ д о
ᴄ 7 ᴍ Ь Ꞵ ȣ Ḫ ẗ ẘ ẙ aʾ ḯ ṷ Ṳ Ṵ Ṷ ᾷ ᾷ ᾇ ᾇ
ᾧ ᾧ ᾯ ᾯ ᾗ ᾗ ᾏ ᾏ ᾟ ᾟ ᾙ ᾙ ᾩ ᾩ ᾫ ᾫ ᾭ ᾭ ᾯ ᾯ
ᾇ ᾇ ᾷ ᾶ ᾷ ̣ ᾐ ᾘ ᾗ ᾖ ᾙ ῒ ΐ ῖ ῗ ῢ ΰ ῤ ῦ ῧ
ᾯ ᾩ ᾫ ᾭ ᾯ D R T ⁂ O i S ⍜ Ⱡ У A J E ᯈ Ꝑ
Ⱥ Ⱶ Г Ꙩ W ძ Ր G M ᖇ ᯅ Ⲟ H Ȝ Ꙗ Ɵ ʯ G Ո h
Z ᖁ Ꝏ I ᔕ ᯒ ᯛ ᯡ Ɛ Ⱨ ᘐ 4 Ь ᡮ ᡏ R Ꮮ Ꮃ Ꮥ Ꮾ
Ꭻ Ꭻ V S ᛕ ஃ Ꮯ L C ᯎ ᯒ P Ꮯ Ᏼ Ꮒ K ძ Ꮯ Ꮆ ᭐
᭐ ᭄ ᭐ 6 ᔕ ff fi fl ffi ffl ft st ⵎⵏⵎⵑⵎⵒⵏⵎⵒ

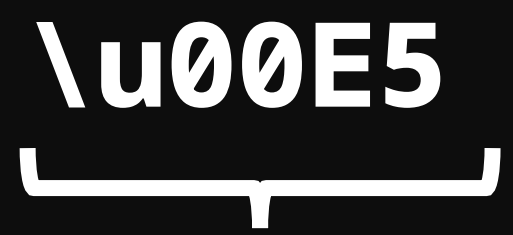

Pålegg


Pålegg
Pålegg
Pålegg
Pålegg

Some strings are equivalent, but different

```
>>> "pålegg" == "pålegg"
False
```

Some strings are equivalent, but different




```
>>> "pålegg" == "pålegg"
False
```

Some strings are equivalent, but different


```
>>> "pålegg" == "pålegg"
False
```

Piñata != Piñata
FJÄLLBO != FJÄLLBO
Beyoncé != Beyoncé

Normalization makes equivalent strings equal

5 Normalization and Case

This section discusses issues that must be taken into account when considering normalization and case folding of identifiers in programming languages or scripting languages. Using normalization avoids many problems where apparently identical identifiers are not treated equivalently. Such problems can appear both during compilation and during linking—in particular across different programming languages. To avoid such problems, programming languages can normalize identifiers before storing or comparing them. Generally if the programming language has case-sensitive identifiers, then Normalization Form C is appropriate; whereas, if the programming language has case-insensitive identifiers, then Normalization Form KC is more appropriate.

Implementations that take normalization and case into account have two choices: to treat variants as equivalent, or to disallow variants.

https://www.unicode.org/reports/tr31/

There are four Unicode normalization forms

	Composed	Decomposed
Canonical	NFC	NFD
Compatibility	NFKC	NFKD

There are four Unicode normalization forms

	Composed	Decomposed
Canonical	NFC	NFD
Compatibility	NFKC	NFKD

The canonical composed form tries to represent the text with as few code-points as possible

```
>>> from unicodedata import
... normalize
>>> len(normalize("NFC", "pålegg"))
6
```

The canonical decomposed form tries to split the “base characters” from their accents

```
>>> from unicodedata import
... normalize
>>> len(normalize("NFD", "pålegg"))
7
```

	Composed	Decomposed
Canonical	NFC	NFD
Compatibility	NFKC	NFKD

Some characters are stylistic subsets of other characters

Symbols:

½ ∰ ⁰⁄₃ ⋘

Superscript:

^{2} ^{5} ^{t} ^{z}

Fancy letters:

ℒ ㏂ 𝒢 𝟟

And over 3800 more!

The compatibility normalizations transform these characters into their unrestricted forms

```
>>> from unicodedata import
... normalize
>>> normalize("NFKC","½")
'1/2'
```

Python uses compatibility normalisation for parsing identifiers

```
>>> 𝕞𝕖ſs𝑎𝒈𝑒 = "Hello world"
>>> 𝕡r𝒾nt(ｍｅ𝕤𝐬𝑎ᵍₑ)
Hello world
```

README

𝔉𝔞𝔫𝔠𝔦𝔣𝔶𝔭𝔶

Are you tired of boring Python code? Or do you want to make a real impression with your next PR? Look no further! 𝔽𝒶𝓃𝒸𝒾𝒻𝓎𝓅𝔂 will transform any plain and boring Python code into a feast for the eyes.

```
pipx run fancifypy code "print(range(5))" --seed 42
```

```
𝔭𝔯𝔦𝔫𝔱 ( 𝔯𝒶𝓃𝑔𝑒 ( 5 ) )
```

Alternatively, you can fancify files directly

```
pipx run fancifypy path script.py
```

and defancified files (if you for some reason want boring files again)

```
pipx run fancifypy boring script.py
```

Note that 𝔽𝒶𝓃𝒸𝒾𝒻𝓎𝓅𝔂 leaves somewhat odd whitespaces, so we recommend using Ruff or Black afterwards to format the code.

Is this safe?

https://github.com/yngvem/fancifypy

“Confusables” are characters that look alike but are different

```
>>> "A" == "A"
False
```

Unicode® Technical Standard #39

UNICODE SECURITY MECHANISMS

Version	16.0.0
Editors	Mark Davis (markdavis@google.com), Michel Suignard (michel@suignard.com)
Date	2024-09-03
This Version	https://www.unicode.org/reports/tr39/tr39-30.html
Previous Version	https://www.unicode.org/reports/tr39/tr39-28.html
Latest Version	https://www.unicode.org/reports/tr39/
Latest Proposed Update	https://www.unicode.org/reports/tr39/proposed.html
Revision	30

Summary

Because Unicode contains such a large number of characters and incorporates the varied writing systems of the world, incorrect usage can expose programs or systems to possible security attacks. This document specifies mechanisms that can be used to detect possible security problems.

Status

This document has been reviewed by Unicode members and other interested parties, and has been approved for publication by the Unicode Consortium. This is a stable document and may be used as reference material or cited as a normative reference by other specifications.

> ***A Unicode Technical Standard (UTS)*** *is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard, see [Unicode]. For a list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

https://www.unicode.org/reports/tr39/
https://www.unicode.org/faq/security.html
https://www.unicode.org/Public/security/16.0.0/intentional.txt
https://www.unicode.org/Public/security/16.0.0/confusables.txt

Some characters require locale-specific handling

```
>>> "I".lower() == "ı"
False
```

Some letters share the same code point

\u014a

Noto sans

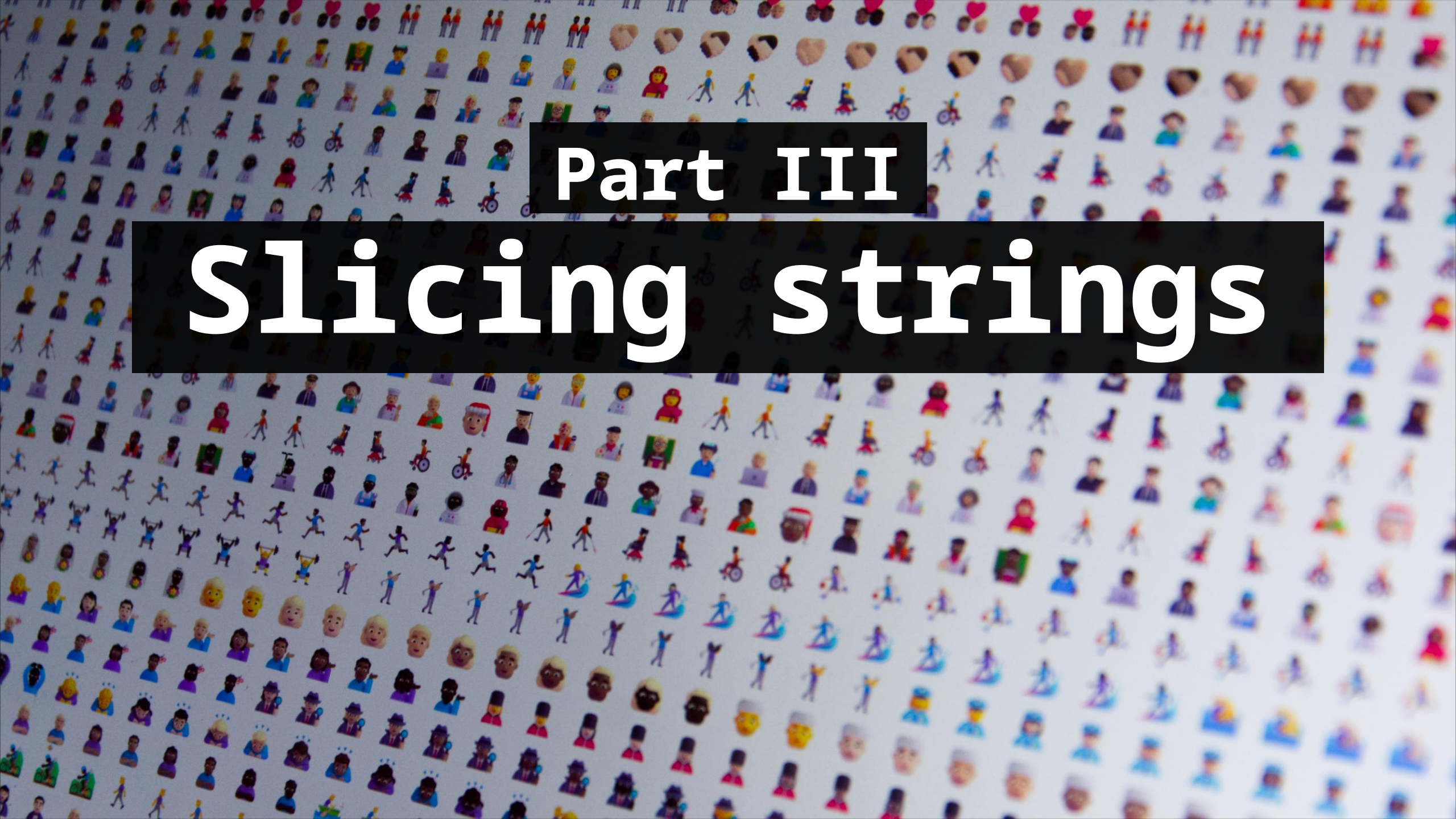
Some letters share the same code point

\u014a

Noto sans

\u014a

Arial


Part III
Slicing strings

```
>>> len("🏳️‍🌈")
```

```
>>> len("🏳️‍🌈")
4
```

```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
```

```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
...
🏳: 1f3f3
```

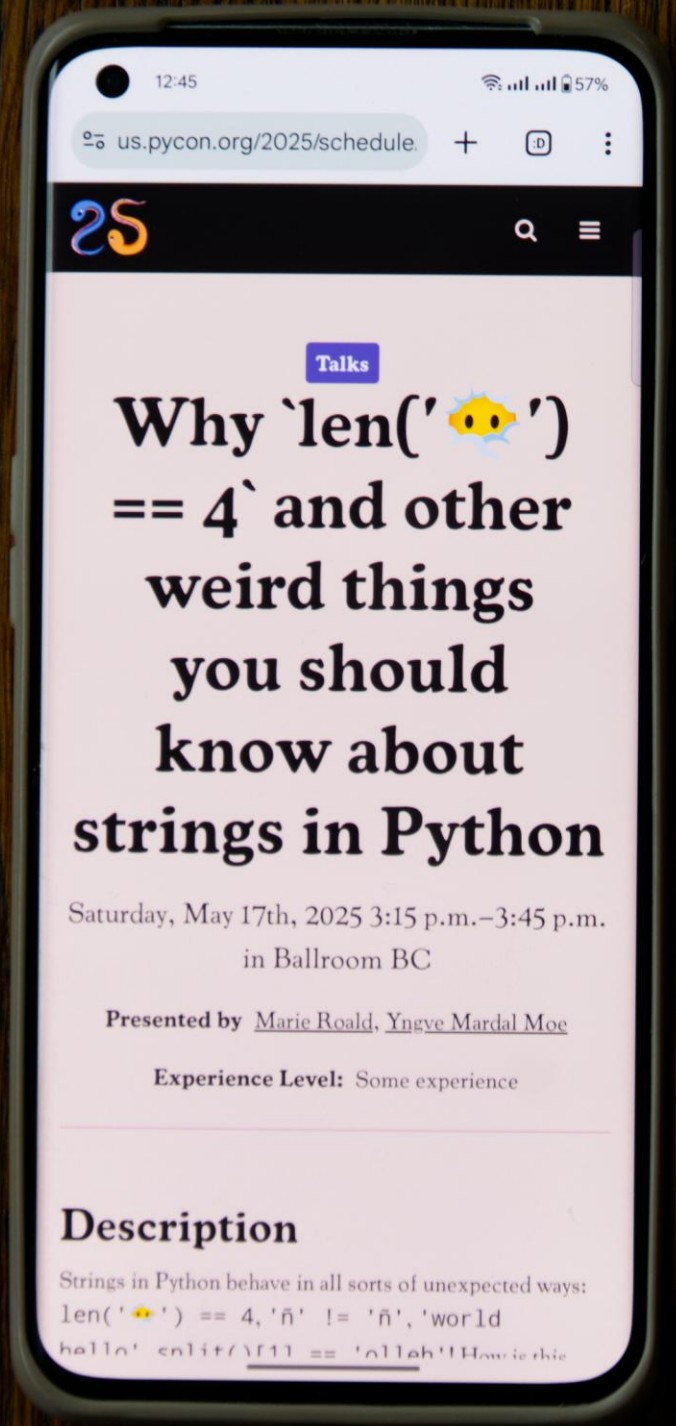
```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
...
🏳: 1f3f3
️: fe0f
‍: 200d
🌈: 1f308
```

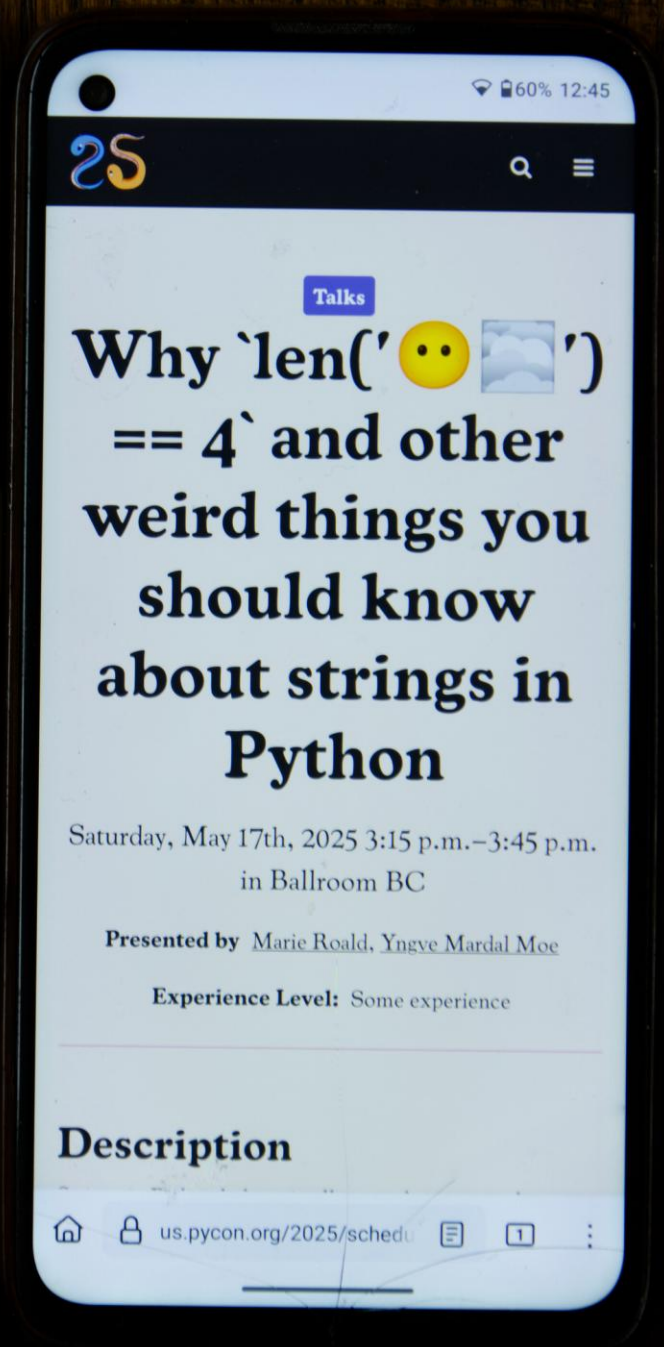
```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
🏳: 1f3f3
️: fe0f
‍: 200d
🌈: 1f308
```

```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
🏳: 1f3f3
️: fe0f
‍: 200d
🌈: 1f308
```

```
>>> print("🏳️‍🌈"[:2])
🏳
```

```
>>> for c in "🏳️‍🌈":
...     print(c, hex(ord(c)))
🏳: 1f3f3
️: fe0f
‍: 200d
🌈: 1f308
```


12:45
57%
us.pycon.org/2025/schedule
Talks
Why `len('🫠')
== 4` and other
weird things
you should
know about
strings in Python
Saturday, May 17th, 2025 3:15 p.m.–3:45 p.m.
in Ballroom BC
Presented by Marie Roald, Yngve Mardal Moe
Experience Level: Some experience
Description
Strings in Python behave in all sorts of unexpected ways:
len('🫠') == 4, 'ñ' != 'ñ', 'world


60% 12:45
Talks
Why `len('😶‍🌫️')
== 4` and other
weird things you
should know
about strings in
Python
Saturday, May 17th, 2025 3:15 p.m.–3:45 p.m.
in Ballroom BC
Presented by Marie Roald, Yngve Mardal Moe
Experience Level: Some experience
Description
us.pycon.org/2025/sched

Accented letters can also form grapheme clusters

```
>>> "pålegg"[:1]
'pa’
```

Accented letters can also form grapheme clusters

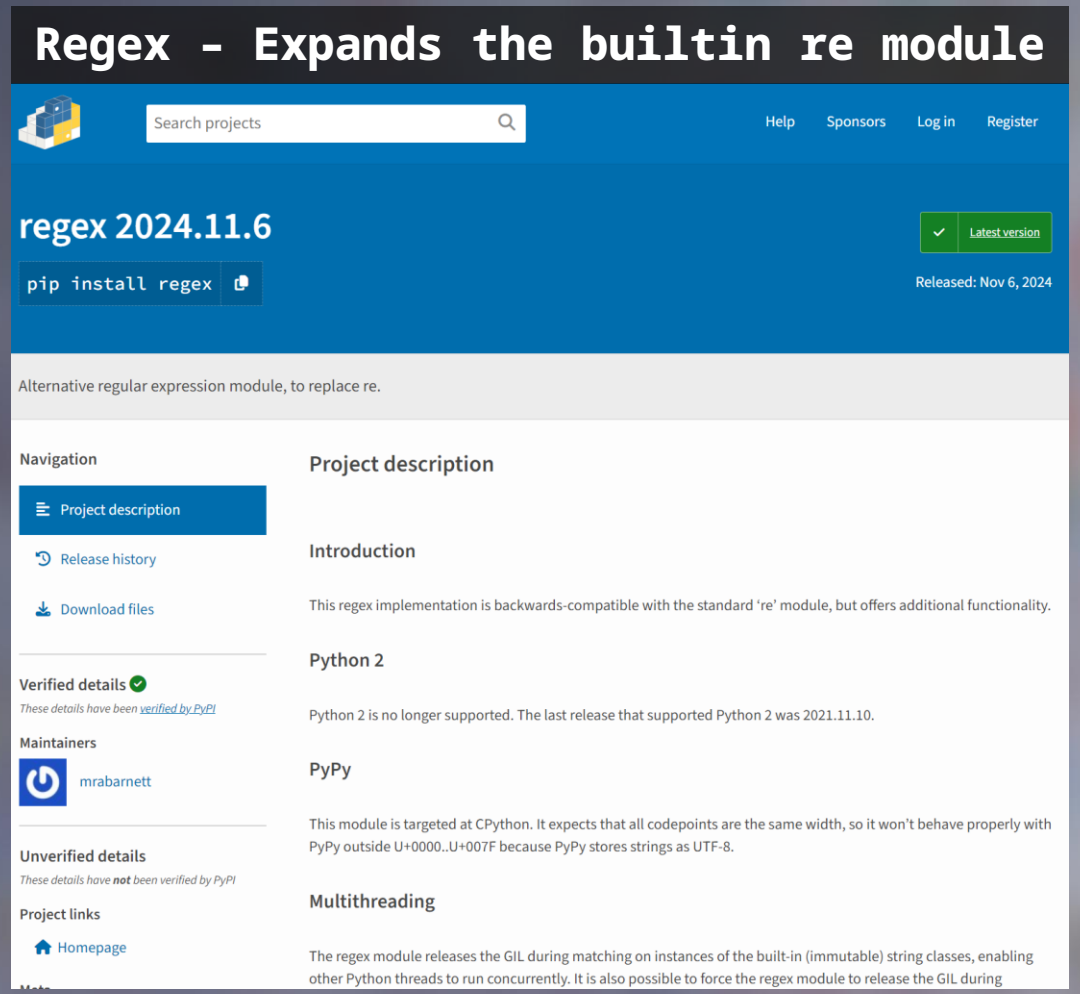
```
>>> "pålegg"[:1]
'pa’

>>> from unicodedata import
... normalize
>>> len(normalize("NFC", "ą"))
2
```

Korean characters can also form grapheme clusters

```
>>> "뱸"[:2]
'뱨'
```

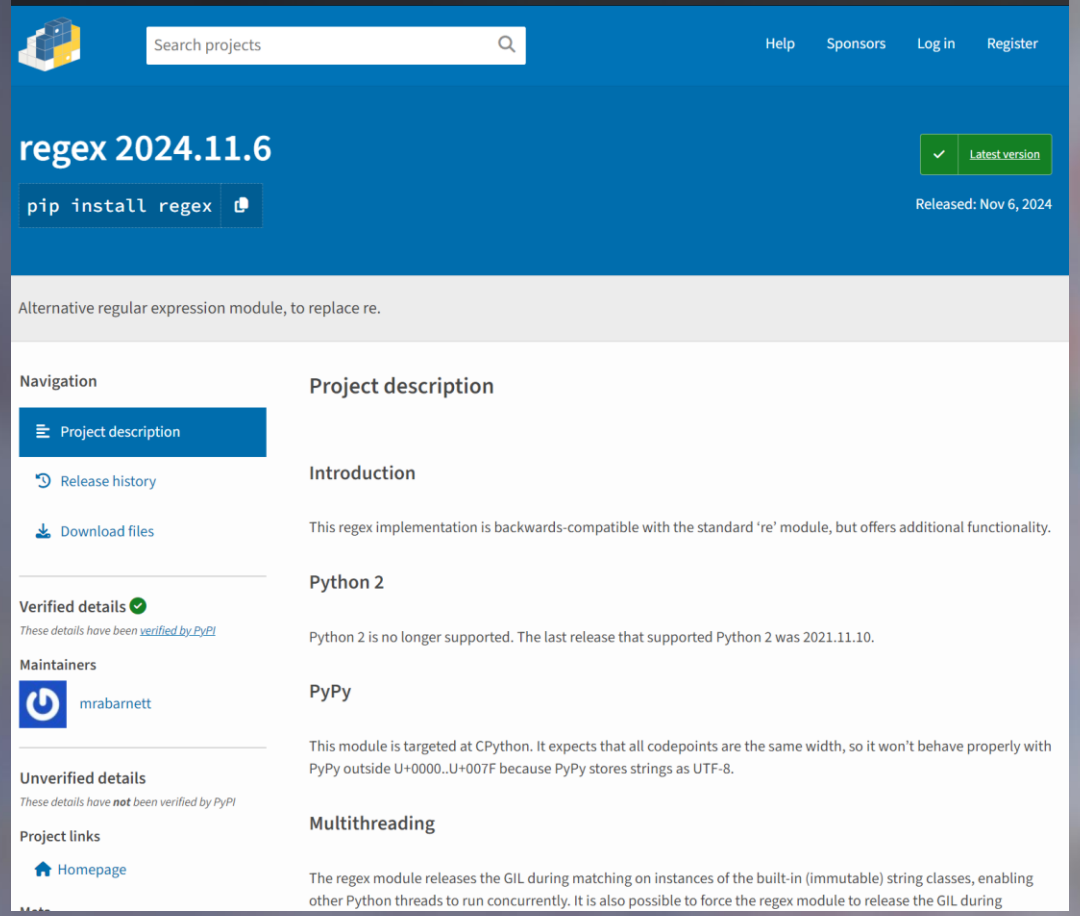
There are several open source tools for better Unicode handling in Python

Regex - Expands the builtin re module

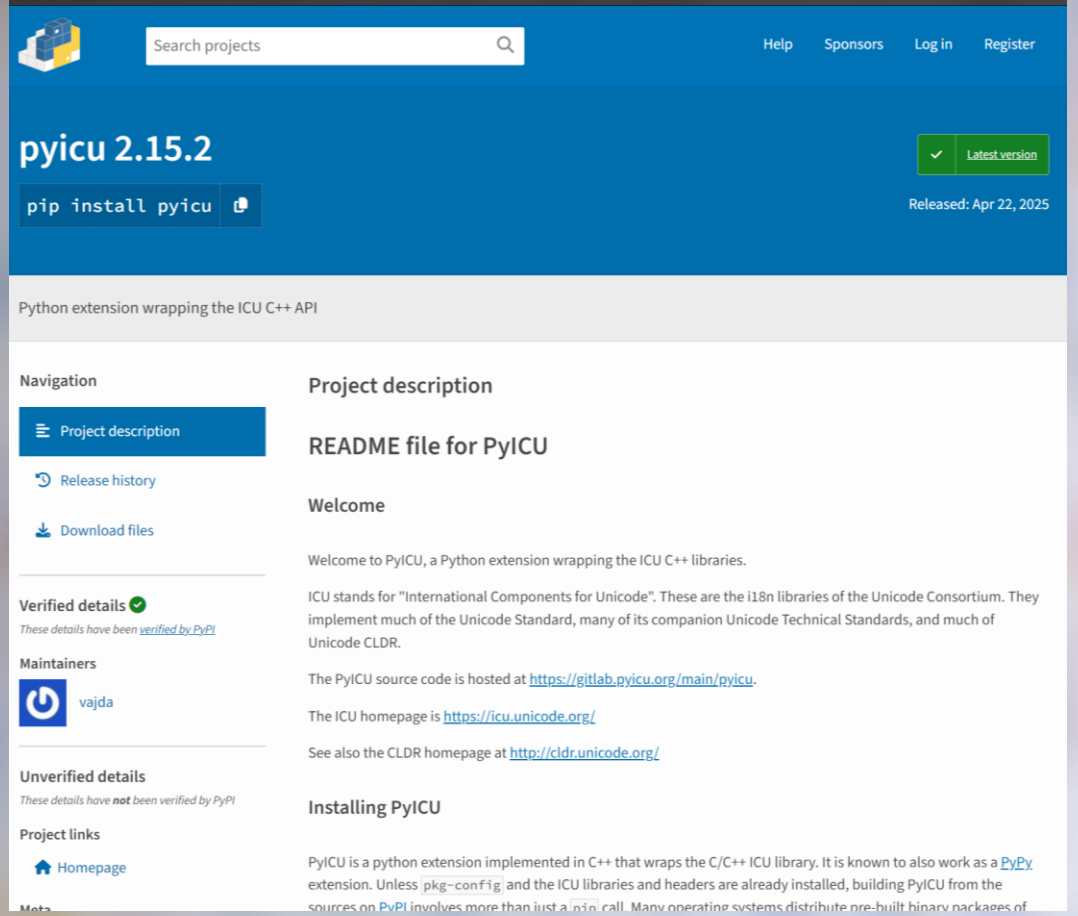


There are several open source tools for better Unicode handling in Python

Regex - Expands the builtin re module



PyICU - Wraps the C/C++ ICU library



You can’t just count grapheme clusters when you measure string length

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🤿 *Casefold for caseless matching*

```
"Weissensee".casefold()
```

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🤿 *Casefold for caseless matching*

```
"Weissensee".casefold()
```

🥪 *Normalise your strings*

```
normalize("NFC", “Pålegg”)
```

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🤿 *Casefold for caseless matching*

```
"Weissensee".casefold()
```

🥪 *Normalise your strings*

```
normalize("NFC", “Pålegg”)
```

🔒 *Consider security implications*

Read Unicode TR39

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🤿 *Casefold for caseless matching*

```
"Weissensee".casefold()
```

🥪 *Normalise your strings*

```
normalize("NFC", “Pålegg”)
```

🔒 *Consider security implications*

Read Unicode TR39

🤯 *Characters are not code points*

Use extended grapheme clusters for visual length

🧀 *Use UTF-8*

```
open(f,encoding="utf-8")
```

🔒 *Consider security implications*

Read Unicode TR39

🤿 *Casefold for caseless matching*

```
"Weissensee".casefold()
```

🤯 *Characters are not code points*

Use extended grapheme clusters for visual length

🥪 *Normalise your strings*

```
normalize("NFC", “Pålegg”)
```

❓ *Unicode is imperfect and not all aspects of Unicode is builtin in Python!*