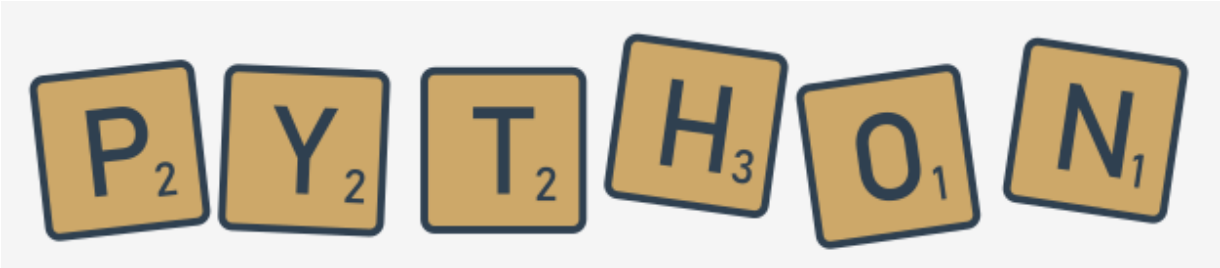
Characters and Strings vs. Computers

How computers understand single characters

You've written some interesting programs since you've started this course, but all of them have processed only one kind of data - numbers. As you know (you can see this everywhere around you) lots of computer data are not numbers: first names, last names, addresses, titles, poems, scientific papers, emails, court judgements, love confessions, and much, much more.



All these data must be stored, input, output, searched, and transformed by contemporary computers just like any other data, no matter if they are single characters or multi-volume encyclopedias.

How is it possible?

How can you do it in Python? This is what we'll discuss now. Let's start with how computers understand single characters.

Computers store characters as numbers. Every character used by a computer corresponds to a unique number, and vice versa. This assignment must include more characters than you might expect. Many of them are invisible to humans, but essential to computers.

Some of these characters are called **whitespaces**, while others are named **control characters**, because their purpose is to control input/output devices.

An example of a whitespace that is completely invisible to the naked eye is a special code, or a pair of codes (different operating systems may treat this issue differently), which are used to mark the ends of the lines inside text files.

People do not see this sign (or these signs), but are able to observe the effect of their application where the lines are broken.

We can create virtually any number of character-number assignments, but life in a world in which every type of computer uses a different character encoding would not be very convenient. This system has led to a need to introduce a universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world.

The one named **ASCII** (short for **American Standard Code for Information Interchange**) is the most widely used, and you can assume that nearly all modern devices (like computers, printers, mobile phones, tablets, etc.) use that code.

The code provides space for **256 different characters**, but we are interested only in the first 128. If you want to see how the code is constructed, look at the table below. Click the table to enlarge it. Look at it carefully - there are some interesting facts. Look at the code of the most common character - the *space*. This is *32*.

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	53	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Now check the code of the lower-case letter *a*. This is *97*. And now find the upper-case *A*. Its code is *65*. Now work out the difference between the code of *a* and *A*. It is equal to *32*. That's the code of a *space*. Interesting, isn't it?

Also note that the letters are arranged in the same order as in the Latin alphabet.

I18N

Of course, the Latin alphabet is not sufficient for the whole of mankind. Users of that alphabet are in the minority. It was necessary to come up with something more flexible and capacious than ASCII, something able to make all the software in the world amenable to **internationalization**, because different languages use completely different alphabets, and sometimes these alphabets are not as simple as the Latin one.

The word *internationalization* is commonly shortened to **I18N**.

Why? Look carefully - there is an *I* at the front of the word, next there are *18* different letters, and an *N* at the end.

Despite the slightly humorous origin, the term is officially used in many documents and standards.

The **software I18N** is a standard in present times. Each program has to be written in a way that enables it to be used all around the world, among different cultures, languages and alphabets.

A classic form of ASCII code uses eight bits for each sign. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters). Is it possible to push all the other national characters used around the world into the remaining 128 locations?

No. It isn't.

Code points and code pages

We need a new term now: a **code point**.

A code point is **a number which makes a character**. For example, *32* is a code point which makes a *space* in ASCII encoding. We can say that standard ASCII code consists of 128 code points.

As standard ASCII occupies 128 out of 256 possible code points, you can only make use of the remaining 128.

It's not enough for all possible languages, but it may be sufficient for one language, or for a small group of similar languages.

Can you **set the higher half of the code points differently for different languages**? Yes, you can. Such a concept is called a **code page**.

A code page is a **standard for using the upper 128 code points to store specific national characters**. For example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages, and so on.

This means that the one and same code point can make different characters when used in different code pages.

For example, the code point *200* makes Č (a letter used by some Slavic languages) when utilized by the ISO/IEC 8859-2 code page, and makes Ш (a Cyrillic letter) when used by the ISO/IEC 8859-5 code page.

In consequence, to determine the meaning of a specific code point, you have to know the target code page.

In other words, the code points derived from code the page concept are ambiguous.

Unicode

Code pages helped the computer industry to solve I18N issues for some time, but it soon turned out that they would not be a permanent solution.

The concept that solved the problem in the long term was **Unicode**.

Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages).

UCS-4

The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes (a group of characters of similar origin, application, or nature).

There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is **UCS-4**.

The name comes from **Universal Character Set**.

UCS-4 uses 32 bits (four bytes) to store each character, and the code is just the Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it.

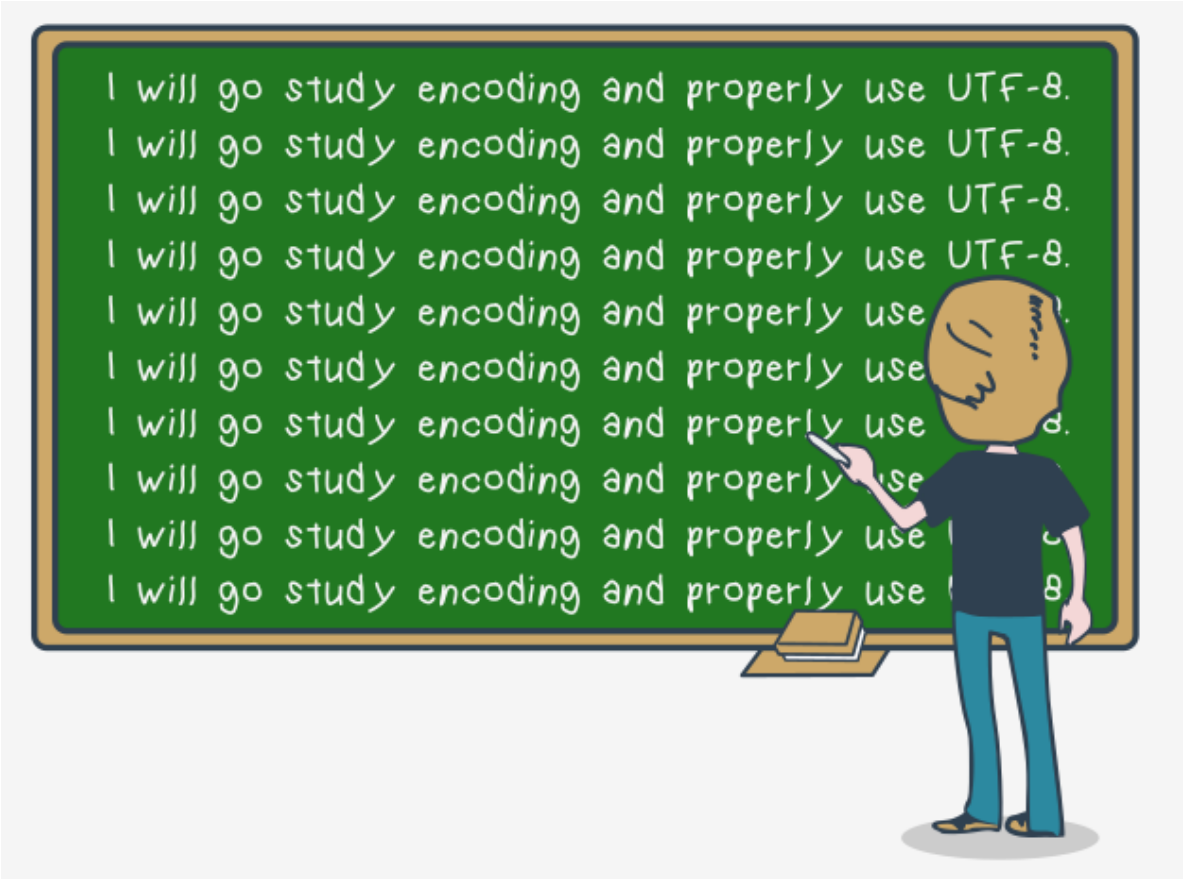
As you can see, UCS-4 is a rather wasteful standard - it increases a text's size by four times compared to standard ASCII. Fortunately, there are smarter forms of encoding Unicode texts.

UTF-8

One of the most commonly used is **UTF-8**.

The name is derived from **Unicode Transformation Format**.

The concept is very smart. **UTF-8 uses as many bits for each of the code points as it really needs to represent them**.



For example:

- all Latin characters (and all standard ASCII characters) occupy eight bits;
- non-Latin characters occupy 16 bits;
- CJK (China-Japan-Korea) ideographs occupy 24 bits.

Due to features of the method used by UTF-8 to store the code points, there is no need to use the BOM, but some of the tools look for it when reading the file, and many editors set it up during the save.

Python 3 fully supports Unicode and UTF-8:

- you can use Unicode/UTF-8 encoded characters to name variables and other entities;
- you can use them during all input and output.

This means that Python3 is completely I18Ned.