Presentation

If I gave you the four bytes
`1010000 1101100 1100001 1101110`

(which are 80, 108, 97, 110 in decimal), what did I mean?

Presentation

If I gave you the four bytes

`1010000 1101100 1100001 1101110`

(which are 80, 108, 97, 110 in decimal), what did I mean?

Is this the encoding of an integer?

Presentation

If I gave you the four bytes
`1010000 1101100 1100001 1101110`

(which are 80, 108, 97, 110 in decimal), what did I mean?

Is this the encoding of an integer?

If so, signed, unsigned, 2s complement?

Presentation

If I gave you the four bytes
`1010000 1101100 1100001 1101110`

(which are 80, 108, 97, 110 in decimal), what did I mean?

Is this the encoding of an integer?

If so, signed, unsigned, 2s complement?

Least significant byte first or most significant byte first?

Presentation

Or is it floating point number?

Presentation

Or is it floating point number?

Or is it a string of four characters?

Presentation

Or is it floating point number?

Or is it a string of four characters?

In ASCII? Or some other encoding like EBCDIC, or UTF-8?

Presentation

From another point of view:

I want to send `"Plan"` to you. What do I send?

Presentation

From another point of view:

I want to send `"Plan"` to you. What do I send?

If we both use ASCII to encode characters, I might send four bytes 80, 108, 97, 110

Presentation

From another point of view:

I want to send `"Plan"` to you. What do I send?

If we both use ASCII to encode characters, I might send four bytes 80, 108, 97, 110

If we both use EBCDIC to encode characters, I might send four bytes 215, 147, 129, 149

Presentation

From another point of view:

I want to send `"Plan"` to you. What do I send?

If we both use ASCII to encode characters, I might send four bytes 80, 108, 97, 110

If we both use EBCDIC to encode characters, I might send four bytes 215, 147, 129, 149

If we use some other encoding, it might need more than four bytes

Presentation

What do I do if we use different encodings? Perhaps my machine uses ASCII while yours uses EBCDIC

Presentation

What do I do if we use different encodings? Perhaps my machine uses ASCII while yours uses EBCDIC

What do I do if I don't know what encoding you use?

Presentation

This is the problem of *presentation*

Presentation

This is the problem of *presentation*

Bits are just bits unless they have some agreed-on meaning

Presentation

This is the problem of *presentation*

Bits are just bits unless they have some agreed-on meaning

And the agreeing is the difficult part

Presentation

This is the problem of *presentation*

Bits are just bits unless they have some agreed-on meaning

And the agreeing is the difficult part

Particularly as some people forget that not everyone uses the same representations for everything

Presentation

The job of the presentation layer is to ensure that the data at one end of a connection is interpreted in the same way when it reaches the other end of the connection

Presentation

The job of the presentation layer is to ensure that the data at one end of a connection is interpreted in the same way when it reaches the other end of the connection

It is about preservation of *meaning*

Presentation

The job of the presentation layer is to ensure that the data at one end of a connection is interpreted in the same way when it reaches the other end of the connection

It is about preservation of *meaning*

So if I send you the number `3.14`, you get the number `3.14`

Presentation

The job of the presentation layer is to ensure that the data at one end of a connection is interpreted in the same way when it reaches the other end of the connection

It is about preservation of *meaning*

So if I send you the number `3.14`, you get the number `3.14`

Even if we use different representations of floating point numbers

Presentation

If I send you the string `"cat"`, you get the string `"cat"`

Presentation

If I send you the string `"cat"`, you get the string `"cat"`

Even if we use different ways of encoding characters

Presentation

If I send you the string `"cat"`, you get the string `"cat"`

Even if we use different ways of encoding characters

Even if we are using different programming languages that encode strings in different ways

Presentation

If I send you a picture containing a particular blue, you get a picture with the same blue

Presentation

If I send you a picture containing a particular blue, you get a picture with the same blue

Even if we are using different representations of pictures

Presentation

If I send you a picture containing a particular blue, you get a picture with the same blue

Even if we are using different representations of pictures

Even if we are using different picture viewers

Presentation

If I send you a picture containing a particular blue, you get a picture with the same blue

Even if we are using different representations of pictures

Even if we are using different picture viewers

Photographers get very wound up about this particular problem!

Presentation

If I send you a picture containing a particular blue, you get a picture with the same blue

Even if we are using different representations of pictures

Even if we are using different picture viewers

Photographers get very wound up about this particular problem!

Exercise Create a plain text (`txt`) file on MacOS or Linux, and view that file on Windows using Notepad. What is happening?

Addendum May 2018: Microsoft has finally fixed this problem, after only 30 years

Presentation

We have many ways of encoding data

Presentation

We have many ways of encoding data

For example, how do we encode the letter ’A’? One popular way is to use a 7 bit number, namely 65

Presentation

We have many ways of encoding data

For example, how do we encode the letter ’A’? One popular way is to use a 7 bit number, namely 65

The *American Standard Code for Information Interchange* (ASCII) is one standard for encoding letters, digits and various punctuation marks

Presentation

We have many ways of encoding data

For example, how do we encode the letter 'A'? One popular way is to use a 7 bit number, namely 65

The *American Standard Code for Information Interchange* (ASCII) is one standard for encoding letters, digits and various punctuation marks

However, it is not the only standard and that is precisely the problem

Presentation

When the Internet began IBM's *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still widely used

Presentation

When the Internet began IBM's *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still widely used

The purpose of EBCDIC is the same as ASCII: encoding characters as numbers

Presentation

When the Internet began IBM’s *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still widely used

The purpose of EBCDIC is the same as ASCII: encoding characters as numbers

The problem is that a file containing the (decimal) byte values

$$80, 108, 97, 110$$

would be interpreted as “Plan” on an ASCII system, but “$\&\%/ >$” on an EBCDIC system

Presentation

When the Internet began IBM’s *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still widely used

The purpose of EBCDIC is the same as ASCII: encoding characters as numbers

The problem is that a file containing the (decimal) byte values

$$80, 108, 97, 110$$

would be interpreted as “Plan” on an ASCII system, but “$\&\%/ >$” on an EBCDIC system

In ASCII, the value 108 means the character ’l’

Presentation

When the Internet began IBM's *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still widely used

The purpose of EBCDIC is the same as ASCII: encoding characters as numbers

The problem is that a file containing the (decimal) byte values

$$80, 108, 97, 110$$

would be interpreted as "Plan" on an ASCII system, but "$\&\%/>$" on an EBCDIC system

In ASCII, the value 108 means the character 'l'

In EBCDIC, the value 108 means the character '%'

Presentation

Philosophy

The *presentation problem* is to ensure that we have the same *meaning* on any system

Presentation

Philosophy

The *presentation problem* is to ensure that we have the same *meaning* on any system

We can easily copy bits from system to system, but our *interpretation* of those bits changes from system to system

Presentation

Philosophy

The *presentation problem* is to ensure that we have the same *meaning* on any system

We can easily copy bits from system to system, but our *interpretation* of those bits changes from system to system

So to make our interpretation consistent we might have to *change the bits*

Presentation

Philosophy

The *presentation problem* is to ensure that we have the same *meaning* on any system

We can easily copy bits from system to system, but our *interpretation* of those bits changes from system to system

So to make our interpretation consistent we might have to *change the bits*

But not only *how* to change them, but *when*

Presentation

Philosophy

If the file 80, 108, 97, 110 is a text file, we must change the values to ensure consistent interpretation

Presentation

Philosophy

If the file 80, 108, 97, 110 is a text file, we must change the values to ensure consistent interpretation

If this is a list of the IQs of four people, we must *not* change the values

Presentation

Philosophy

If the file 80, 108, 97, 110 is a text file, we must change the values to ensure consistent interpretation

If this is a list of the IQs of four people, we must *not* change the values

Everything depends on the final interpretation of the data: this is a subtle point and is why presentation issues are often ignored or incorrectly implemented

Presentation

Note that IP *does not address presentation*, and leaves it to the application

Presentation

Note that IP *does not address presentation*, and leaves it to the application

This means that presentation must be addressed by the programmer in their all their applications

Presentation

Note that IP *does not address presentation*, and leaves it to the application

This means that presentation must be addressed by the programmer in their all their applications

In the early Internet all the machine were the same, so presentation was not realised to be a problem

Presentation

Note that IP *does not address presentation*, and leaves it to the application

This means that presentation must be addressed by the programmer in their all their applications

In the early Internet all the machine were the same, so presentation was not realised to be a problem

Today, things are very different

Presentation

Note that IP *does not address presentation*, and leaves it to the application

This means that presentation must be addressed by the programmer in their all their applications

In the early Internet all the machine were the same, so presentation was not realised to be a problem

Today, things are very different

And programmers are still forgetting this is an issue

Presentation

These days most people have more-or-less settled on ASCII as the encoding to use for simple Latin/Roman letters and digits

Presentation

These days most people have more-or-less settled on ASCII as the encoding to use for simple Latin/Roman letters and digits

So presentation issues are minimal for these kinds of text data

Presentation

These days most people have more-or-less settled on ASCII as the encoding to use for simple Latin/Roman letters and digits

So presentation issues are minimal for these kinds of text data

On the other hand, other character sets (Chinese, Russian, Klingon, etc.) are in the ascendant, with the *Universal Coded Character Set* (UCS) plus *Unicode* being the chosen representation

Presentation

UCS/Unicode

UCS (ISO 10646) is a character encoding that uses 31 bits instead of just 7

Presentation

UCS/Unicode

UCS (ISO 10646) is a character encoding that uses 31 bits instead of just 7

This gives ample room for all the characters in all the written languages in the world

Presentation

UCS/Unicode

UCS (ISO 10646) is a character encoding that uses 31 bits instead of just 7

This gives ample room for all the characters in all the written languages in the world

It is a big table that says “this value represents this character”

Presentation

UCS/Unicode

UCS (ISO 10646) is a character encoding that uses 31 bits instead of just 7

This gives ample room for all the characters in all the written languages in the world

It is a big table that says “this value represents this character”

Unicode takes UCS and adds details like direction of writing (left-to-right or right-to-left or bidirectional), defining alphabetic orders, which are capital letters, and so on

Presentation

UCS/Unicode

Unicode only uses UCS values from 0 to 10FFFF

Presentation

UCS/Unicode

Unicode only uses UCS values from 0 to 10FFFF

A maximum of $17 \times 2^{16} = 1,114,112$ *code points*

Presentation

UCS/Unicode

Unicode only uses UCS values from 0 to 10FFFF

A maximum of $17 \times 2^{16} = 1,114,112$ *code points*

A code point can denote a character or a character modifier, e.g., a variant or a combining character like an accent

Presentation

UCS/Unicode

Unicode only uses UCS values from 0 to 10FFFF

A maximum of $17 \times 2^{16} = 1,114,112$ *code points*

A code point can denote a character or a character modifier, e.g., a variant or a combining character like an accent

For example, é is a single character, while é is two code points: e followed by a combining character ’

Presentation

UCS/Unicode

Unicode only uses UCS values from 0 to 10FFFF

A maximum of $17 \times 2^{16} = 1,114,112$ *code points*

A code point can denote a character or a character modifier, e.g., a variant or a combining character like an accent

For example, é is a single character, while é is two code points: e followed by a combining character ’

2,048 code points are excluded (the surrogate values D800–DFFF for backwards compatability with UTF-16, below), so the number of representable characters (more properly: *graphemes*) is just 1,112,064

Presentation

UCS/Unicode

And then there is the *glyph*, the visible rendering of the grapheme in some font: é and é

Presentation

UCS/Unicode

And then there is the *glyph*, the visible rendering of the grapheme in some font: é and é

Code points can be written as “U+hex”, e.g., U+C2A3 for the index of code point '£')

Presentation

UCS/Unicode

But using 4 bytes per character would not be appreciated by many programmers since it would

Presentation

UCS/Unicode

But using 4 bytes per character would not be appreciated by many programmers since it would

- break the “one character is one byte” assumption many programs make

Presentation

UCS/Unicode

But using 4 bytes per character would not be appreciated by many programmers since it would

- break the “one character is one byte” assumption many programs make
- make data files four times as large when the original data were encoded in ASCII, and

Presentation

UCS/Unicode

But using 4 bytes per character would not be appreciated by many programmers since it would

- break the “one character is one byte” assumption many programs make
- make data files four times as large when the original data were encoded in ASCII, and
- the zero byte is often conventionally used to mean “end of string” so a value such as (hex) 12 34 00 78 is open to misinterpretation

Presentation

UCS/Unicode

So some *encoding systems* are defined: they implement UCS but use differing numbers of bytes to encode the index into its big table of characters

Presentation

UCS/Unicode

So some *encoding systems* are defined: they implement UCS but use differing numbers of bytes to encode the index into its big table of characters

Some systems are backwardly compatible with ASCII in the sense that values 00 to 7f are the same as their ASCII equivalents

Presentation

UCS/Unicode

So some *encoding systems* are defined: they implement UCS but use differing numbers of bytes to encode the index into its big table of characters

Some systems are backwardly compatible with ASCII in the sense that values 00 to 7f are the same as their ASCII equivalents

The simplest method, *Unicode Transformation Format* 32 (UTF-32, also called UCS-4), simply uses four bytes per character and embeds ASCII in UCS by merely adding three 0 bytes before every ASCII byte

Presentation

UCS/Unicode

`tiger` in ASCII is five bytes: 116 106 103 101 114 `tiger` in ASCII is five bytes: 116 106 103 101 114

`tiger` in UTF-32 is 20 bytes: 0 0 0 116 0 0 0 106 0 0 0 103 0 0 0 101 0 0 0 114

Presentation

UCS/Unicode

`tiger` in ASCII is five bytes: 116 106 103 101 114

`tiger` in UTF-32 is 20 bytes: 0 0 0 116 0 0 0 106 0 0 0 103 0 0 0 101 0 0 0 114

老虎 is 0 0 128 1 0 0 134 78

Presentation

UCS/Unicode

`tiger` in ASCII is five bytes: 116 106 103 101 114

`tiger` in UTF-32 is 20 bytes: 0 0 0 116 0 0 0 106 0 0 0 103 0 0 0 101 0 0 0 114

老虎 is 0 0 128 1 0 0 134 78

This has the expansion and zero problems

Presentation

UCS/Unicode

`tiger` in ASCII is five bytes: 116 106 103 101 114

`tiger` in UTF-32 is 20 bytes: 0 0 0 116 0 0 0 106 0 0 0 103 0 0 0 101 0 0 0 114

老虎 is 0 0 128 1 0 0 134 78

This has the expansion and zero problems

But is convenient if we are working with individual characters (rather than strings) as 32 bit values

Presentation

UCS/Unicode

`tiger` in ASCII is five bytes: 116 106 103 101 114

`tiger` in UTF-32 is 20 bytes: 0 0 0 116 0 0 0 106 0 0 0 103 0 0 0 101 0 0 0 114

老虎 is 0 0 128 1 0 0 134 78

This has the expansion and zero problems

But is convenient if we are working with individual characters (rather than strings) as 32 bit values

For example, indexing into an array of characters is very easy: exactly like indexing into an array of 32-bit integers

Presentation

UCS/Unicode

Less inflationary is UCS-2, that uses *two bytes* per character and prepends a single 0 byte before each ASCII character

Presentation

UCS/Unicode

Less inflationary is UCS-2, that uses *two bytes* per character and prepends a single 0 byte before each ASCII character

This only doubles the size of an ASCII file

Presentation

UCS/Unicode

Less inflationary is UCS-2, that uses *two bytes* per character and prepends a single 0 byte before each ASCII character

This only doubles the size of an ASCII file

Still has the zero problem

UCS-2 was devised for an earlier 16 bit coding (now called the *Basic Multilingual Plane*, or BMP), that was soon found to be too small (not enough characters)

Presentation

UCS/Unicode

UCS-2 was devised for an earlier 16 bit coding (now called the *Basic Multilingual Plane*, or BMP), that was soon found to be too small (not enough characters)

UCS-2 can't represent all possible UCS values. Not even all Unicode values

Presentation

UCS/Unicode

UCS-2 was devised for an earlier 16 bit coding (now called the *Basic Multilingual Plane*, or BMP), that was soon found to be too small (not enough characters)

UCS-2 can't represent all possible UCS values. Not even all Unicode values

Thus the need for UTF-16 which uses *pairs* of UCS-2 values to extend the encoding range

Presentation

UCS/Unicode

UCS-2 was devised for an earlier 16 bit coding (now called the *Basic Multilingual Plane*, or BMP), that was soon found to be too small (not enough characters)

UCS-2 can't represent all possible UCS values. Not even all Unicode values

Thus the need for UTF-16 which uses *pairs* of UCS-2 values to extend the encoding range

UTF-16 can represent all Unicode values, but at the cost of some complexity

Presentation

UCS/Unicode

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Presentation

UCS/Unicode

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

- Get 10 high bits from $x -$ D800

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

- Get 10 high bits from $x -$ D800
- Get 10 low bits from $y -$ DC00

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

- Get 10 high bits from x − D800
- Get 10 low bits from y − DC00
- Concatenate these bits to get a 20 bit value

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

- Get 10 high bits from x − D800
- Get 10 low bits from y − DC00
- Concatenate these bits to get a 20 bit value
- Add hex 10000 to get the UCS value

It uses pairs of 16 bit values in the range D800 to DFFF (*surrogate pairs*) to encode the extended values

Given a pair of surrogate values x in the range D800-DBFF and y in the range DC00-DFFF

- Get 10 high bits from $x - \text{D800}$
- Get 10 low bits from $y - \text{DC00}$
- Concatenate these bits to get a 20 bit value
- Add hex 10000 to get the UCS value

Exercise Compare this to byte stuffing

Presentation

UCS/Unicode

The surrogate values (and which is high and low) can easily be identified in a byte stream: important if you are dipping into the middle of a string

Presentation

UCS/Unicode

The surrogate values (and which is high and low) can easily be identified in a byte stream: important if you are dipping into the middle of a string

It does punch a hole in Unicode from D800 to DFFF that can’t be used as characters

Presentation

UCS/Unicode

The surrogate values (and which is high and low) can easily be identified in a byte stream: important if you are dipping into the middle of a string

It does punch a hole in Unicode from D800 to DFFF that can't be used as characters

The Unicode consortium guarantees never to allocate characters in that range

Presentation

UCS/Unicode

The surrogate values (and which is high and low) can easily be identified in a byte stream: important if you are dipping into the middle of a string

It does punch a hole in Unicode from D800 to DFFF that can't be used as characters

The Unicode consortium guarantees never to allocate characters in that range

UTF-16 is quite popular in use, e.g., Java, C# and various versions of the Windows OS use it for their internal representations of strings

Presentation

UCS/Unicode

The most important representation, UTF-8, represents all ASCII (7 bit) values as themselves while still being able to represent the full UCS range

Presentation

UCS/Unicode

The most important representation, UTF-8, represents all ASCII (7 bit) values as themselves while still being able to represent the full UCS range

UCS values 00000000 to 0000007F are encoded as single bytes 00 to 7f. Thus an ASCII file is a valid UTF-8 file

Presentation

UCS/Unicode

The most important representation, UTF-8, represents all ASCII (7 bit) values as themselves while still being able to represent the full UCS range

UCS values 00000000 to 0000007F are encoded as single bytes 00 to 7f. Thus an ASCII file is a valid UTF-8 file

So, for example, the byte 3F in UTF-8-encoded a file encodes for UCS index 0000003F

Presentation

UCS/Unicode

The most important representation, UTF-8, represents all ASCII (7 bit) values as themselves while still being able to represent the full UCS range

UCS values 00000000 to 0000007F are encoded as single bytes 00 to 7f. Thus an ASCII file is a valid UTF-8 file

So, for example, the byte 3F in UTF-8-encoded a file encodes for UCS index 0000003F

UCS values 00000080 to 000007FF become two bytes 110xxxxx 10xxxxxx. The last 11 bits from the UCS values are copied across

Presentation

UCS/Unicode

So '£', UCS 000000A3, binary

00000000 00000000 00000000 10100011

becomes 11000010 10100011 (C2A3), since

$$00010/100011 \rightarrow 110/0010\ 10/100011$$

Presentation

UCS/Unicode

So '£', UCS 000000A3, binary

$$00000000\ 00000000\ 00000000\ 10100011$$

becomes 11000010 10100011 (C2A3), since

$$00010/100011 \rightarrow 110/0010\ 10/100011$$

And thus the *two* bytes C2A3 in a file encode the UCS index 000000A3

Presentation

UCS/Unicode

Generally we can encode:

UCS range (hex)	Encoding (binary)
00000000-0000007F	0xxxxxxx
00000080-000007FF	110xxxxx 10xxxxxx
00000800-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000-001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000-03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000-7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Presentation

UCS/Unicode

This table is more than UTF-8 requires

This table is more than UTF-8 requires

The UTF-8 encoding is only defined for values up 10FFFF, for compatibility with Unicode and UTF-16

Presentation

UCS/Unicode

This table is more than UTF-8 requires

The UTF-8 encoding is only defined for values up 10FFFF, for compatibility with Unicode and UTF-16

So only the first four rows of the table

Presentation

UCS/Unicode

Unicode range (hex)	Encoding (binary)
00000000-0000007F	0xxxxxxx
00000080-000007FF	110xxxxx 10xxxxxx
00000800-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Presentation

UCS/Unicode

A full 31-bit range would require up to 6 bytes to encode

Presentation

UCS/Unicode

A full 31-bit range would require up to 6 bytes to encode

Unicode will need at most four (and so will fit in a 32 bit `int`)

A full 31-bit range would require up to 6 bytes to encode

Unicode will need at most four (and so will fit in a 32 bit `int`)

Most common characters only require three or fewer; a majority in use need two or fewer

Presentation

UCS/Unicode

A full 31-bit range would require up to 6 bytes to encode

Unicode will need at most four (and so will fit in a 32 bit `int`)

Most common characters only require three or fewer; a majority in use need two or fewer

And ASCII values only require one byte

Presentation

UCS/Unicode

A full 31-bit range would require up to 6 bytes to encode

Unicode will need at most four (and so will fit in a 32 bit `int`)

Most common characters only require three or fewer; a majority in use need two or fewer

And ASCII values only require one byte

An ASCII file is already a UTF-8 file and there is no expansion of data when regarding it as UCS

Presentation

UCS/Unicode

- ASCII values represent themselves

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works
- The length of each non-ASCII character is given by the number of leading 1 bits

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works
- The length of each non-ASCII character is given by the number of leading 1 bits
- When dipping at random into a UTF-8 encoded file it is easy to find the start of the next character: just search until you find a byte starting with bits 0 or 11

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works
- The length of each non-ASCII character is given by the number of leading 1 bits
- When dipping at random into a UTF-8 encoded file it is easy to find the start of the next character: just search until you find a byte starting with bits 0 or 11
- In the same way, if a byte is lost (e.g., discarded as corrupt) it is easy to re-synchronise

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works
- The length of each non-ASCII character is given by the number of leading 1 bits
- When dipping at random into a UTF-8 encoded file it is easy to find the start of the next character: just search until you find a byte starting with bits 0 or 11
- In the same way, if a byte is lost (e.g., discarded as corrupt) it is easy to re-synchronise
- All UCS values can be encoded

Presentation

UCS/Unicode

- ASCII values represent themselves
- No ASCII character appears as a sub-part of any other character
- The convention of using 0 as end of string still works
- The length of each non-ASCII character is given by the number of leading 1 bits
- When dipping at random into a UTF-8 encoded file it is easy to find the start of the next character: just search until you find a byte starting with bits 0 or 11
- In the same way, if a byte is lost (e.g., discarded as corrupt) it is easy to re-synchronise
- All UCS values can be encoded
- The comparison order of UCS is preserved

Presentation

UCS/Unicode

- UTF-16 does not preserve UCS comparison order

Presentation

UCS/Unicode

- UTF-16 does not preserve UCS comparison order
- both UTF-8 and UTF-16 need up to four bytes to represent Unicode values

Presentation

UCS/Unicode

- UTF-16 does not preserve UCS comparison order
- both UTF-8 and UTF-16 need up to four bytes to represent Unicode values
- UTF-8 is byte order independent

Presentation

UCS/Unicode

- UTF-16 does not preserve UCS comparison order
- both UTF-8 and UTF-16 need up to four bytes to represent Unicode values
- UTF-8 is byte order independent
- UTF-16 comes in big (UTF-16BE) and little endian (UTF-16LE) variants as well as plain UTF-16, when files can employ a special *byte order mark* (BOM, U+FEFF) at their start to establish order

Presentation

UCS/Unicode

- UTF-16 does not preserve UCS comparison order
- both UTF-8 and UTF-16 need up to four bytes to represent Unicode values
- UTF-8 is byte order independent
- UTF-16 comes in big (UTF-16BE) and little endian (UTF-16LE) variants as well as plain UTF-16, when files can employ a special *byte order mark* (BOM, U+FEFF) at their start to establish order
- UTF-32 is big endian

Presentation

UCS/Unicode

- UTF-8 is more efficient on Western character sets; UTF-16 is more efficient on Asian character sets (note that most computer code is written in ASCII)

Presentation

UCS/Unicode

- UTF-8 is more efficient on Western character sets; UTF-16 is more efficient on Asian character sets (note that most computer code is written in ASCII)
- As they are variable length encodings, neither UTF-8 nor UTF-16 allow indexing directly into a string

Presentation

UCS/Unicode

- UTF-8 is more efficient on Western character sets; UTF-16 is more efficient on Asian character sets (note that most computer code is written in ASCII)
- As they are variable length encodings, neither UTF-8 nor UTF-16 allow indexing directly into a string

The advantages of UTF-8 are such that UTF-16 should be retired, but this may take some time

Exercise Have a look at how (or if) your favourite programming language supports UCS or Unicode. E.g., C programmers have `wchar_t`

Exercise A typical programming language has variables syntax that “start with a letter, then letters and digits”. How would this work in Unicode?

Exercise Read about the *Punycode* encoding

Exercise Unicode is split into 17 *planes*. Read about this