Unicode Regular Expressions

Unicode is a character set that aims to define all characters and glyphs from all human languages, living and dead. With more and more software being required to support multiple languages, or even just *any* language, Unicode has been strongly gaining popularity in recent years. Using different character sets for different languages is simply too cumbersome for programmers and users.

Unfortunately, Unicode brings its own requirements and pitfalls when it comes to regular expressions. Of the regex flavors discussed in this tutorial, Java, XML and the .NET framework use Unicode-based regex engines. Perl supports Unicode starting with version 5.6. PCRE can optionally be compiled with Unicode support. Note that PCRE is far less flexible in what it allows for the `\p` tokens, despite its name "Perl-compatible". The PHP preg functions, which are based on PCRE, support Unicode when the /u option is appended to the regular expression. Ruby supports Unicode escapes and properties in regular expressions starting with version 1.9. XRegExp brings support for Unicode properties to JavaScript.

RegexBuddy's regex engine is fully Unicode-based starting with version 2.0.0. RegexBuddy 1.x.x did not support Unicode at all. PowerGREP uses the same Unicode regex engine starting with version 3.0.0. Earlier versions would convert Unicode files to ANSI prior to grepping with an 8-bit (i.e. non-Unicode) regex engine. EditPad Pro supports Unicode starting with version 6.0.0.

Characters, Code Points, and Graphemes or How Unicode Makes a Mess of Things

Most people would consider à a single character. Unfortunately, it need not be depending on the meaning of the word "character".

All Unicode regex engines discussed in this tutorial treat any single Unicode *code point* as a single character. When this tutorial tells you that the dot matches any single character, this translates into Unicode parlance as "the dot matches any single Unicode code point". In Unicode, à can be encoded as two code points: U+0061 (a) followed by U+0300 (grave accent). In this situation, `.` applied to à will match a without the accent. `^.$` will fail to match, since the string consists of two code points. `^..$` matches à.

The Unicode code point U+0300 (grave accent) is a *combining mark*. Any code point that is not a combining mark can be followed by any number of combining marks. This sequence, like U+0061 U+0300 above, is displayed as a single *grapheme* on the screen.

Unfortunately, à can also be encoded with the single Unicode code point U+00E0 (a with grave accent). The reason for this duality is that many historical character sets encode "a with grave accent" as a single character. Unicode's designers thought it would be useful to have a one-on-one mapping with popular legacy character sets, in addition to the Unicode way of separating marks and base letters (which makes arbitrary combinations not supported by legacy character sets possible).

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it's encoded as a single code point, or as multiple code points using combining marks, is easy in Perl, PCRE, PHP, Boost, Ruby 2.0, Java 9, and the Just Great Software applications: simply use `\X`. You can consider `\X` the Unicode version of the dot. There is one difference, though: `\X` always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

In .NET, Java 8 and prior, and Ruby 1.9 you can use `\P{M}\p{M}*+` or `(?>\P{M}\p{M}*)` as a reasonably close substitute. To match any number of graphemes, use `(?>\P{M}\p{M}*)+` as a substitute for `\X+`.

Matching a Specific Code Point

To match a specific Unicode code point, use `\uFFFF` where FFFF is the hexadecimal number of the code point you want to match. You must always specify 4 hexadecimal digits E.g. `\u00E0` matches à, but only when encoded as a single code point U+00E0.

Perl, PCRE, Boost, and std::regex do not support the `\uFFFF` syntax. They use `\x{FFFF}` instead. You can omit leading zeros in the hexadecimal number between the curly braces. Since \x by itself is not a valid regex token, `\x{1234}` can never be confused to match \x 1234 times. It always matches the Unicode code point U+1234. `\x{1234}{5678}` will try to match code point U+1234 exactly 5678 times.

In Java, the regex token `\uFFFF` only matches the specified code point, even when you turned on canonical equivalence. However, the same syntax `\uFFFF` is also used to insert Unicode characters into literal strings in the Java source code. `Pattern.compile("\u00E0")` will match both the single-code-point and double-code-point encodings of à, while `Pattern.compile("\\u00E0")` matches only the single-code-point version. Remember that when writing a regex as a Java string literal, backslashes must be escaped. The former Java code compiles the regex à, while the latter compiles `\u00E0`. Depending on what you're doing, the difference may be significant.

JavaScript, which does not offer any Unicode support through its RegExp class, does support `\uFFFF` for matching a single Unicode code point as part of its string syntax.

XML Schema and XPath do not have a regex token for matching Unicode code points. However, you can easily use XML entities like `&#xFFFF;` to insert literal code points into your regular expression.

Unicode Categories

In addition to complications, Unicode also brings new possibilities. One is that each Unicode character belongs to a certain category. You can match a single character belonging to the "letter" category with `\p{L}`. You can match a single character *not* belonging to that category with `\P{L}`.

Again, "character" really means "Unicode code point". `\p{L}` matches a single code point in the category "letter". If your input string is à encoded as U+0061 U+0300, it matches a without the accent. If the input is à encoded as U+00E0, it matches à with the accent. The reason is that both the code points U+0061 (a) and U+00E0 (à) are in the category "letter", while U+0300 is in the category "mark".

You should now understand why `\P{M}\p{M}*+` is the equivalent of `\X`. `\P{M}` matches a code point that is not a combining mark, while `\p{M}*+` matches zero or more code points that are combining marks. To match a letter including any diacritics, use `\p{L}\p{M}*+`. This last regex will always match à, regardless of how it is encoded. The possessive quantifier makes sure that backtracking doesn't cause `\P{M}\p{M}*+` to match a non-mark without the combining marks that follow it, which `\X` would never do.

PCRE, PHP, and .NET are case sensitive when it checks the part between curly braces of a \p token. `\p{Zs}` will match any kind of space character, while `\p{zs}` will throw an error. All other regex engines described in this tutorial will match the space in both cases, ignoring the case of the category between the curly braces. Still, I recommend you make a habit of using the same uppercase and lowercase combination as I did in the list of properties below. This will make your regular expressions work with all Unicode regex engines.

In addition to the standard notation, `\p{L}`, Java, Perl, PCRE, the JGsoft engine, and XRegExp 3 allow you to use the shorthand `\pL`. The shorthand only works with single-letter Unicode properties. `\pLl` is *not* the equivalent of `\p{Ll}`. It is the equivalent of `\p{L}l` which matches `Al` or `àl` or any Unicode letter followed by a literal `l`.

Perl, XRegExp, and the JGsoft engine also support the longhand `\p{Letter}`. You can find a complete list of all Unicode properties below. You may omit the underscores or use hyphens or spaces instead.

- `\p{L}` or `\p{Letter}`: any kind of letter from any language.
  - `\p{Ll}` or `\p{Lowercase_Letter}`: a lowercase letter that has an uppercase variant.
  - `\p{Lu}` or `\p{Uppercase_Letter}`: an uppercase letter that has a lowercase variant.
  - `\p{Lt}` or `\p{Titlecase_Letter}`: a letter that appears at the start of a word when only the first letter of the word is capitalized.
  - `\p{L&}` or `\p{Cased_Letter}`: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
  - `\p{Lm}` or `\p{Modifier_Letter}`: a special character that is used like a letter.
  - `\p{Lo}` or `\p{Other_Letter}`: a letter or ideograph that does not have lowercase and uppercase variants.
- `\p{M}` or `\p{Mark}`: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
  - `\p{Mn}` or `\p{Non_Spacing_Mark}`: a character intended to be combined with another character without taking up extra space (e.g. accents, umlauts, etc.).
  - `\p{Mc}` or `\p{Spacing_Combining_Mark}`: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
  - `\p{Me}` or `\p{Enclosing_Mark}`: a character that encloses the character it is combined with (circle, square, keycap, etc.).
- `\p{Z}` or `\p{Separator}`: any kind of whitespace or invisible separator.
  - `\p{Zs}` or `\p{Space_Separator}`: a whitespace character that is invisible, but does take up space.
  - `\p{Zl}` or `\p{Line_Separator}`: line separator character U+2028.
  - `\p{Zp}` or `\p{Paragraph_Separator}`: paragraph separator character U+2029.
- `\p{S}` or `\p{Symbol}`: math symbols, currency signs, dingbats, box-drawing characters, etc.
  - `\p{Sm}` or `\p{Math_Symbol}`: any mathematical symbol.
  - `\p{Sc}` or `\p{Currency_Symbol}`: any currency sign.
  - `\p{Sk}` or `\p{Modifier_Symbol}`: a combining character (mark) as a full character on its own.
  - `\p{So}` or `\p{Other_Symbol}`: various symbols that are not math symbols, currency signs, or combining characters.
- `\p{N}` or `\p{Number}`: any kind of numeric character in any script.
  - `\p{Nd}` or `\p{Decimal_Digit_Number}`: a digit zero through nine in any script except ideographic scripts.
  - `\p{Nl}` or `\p{Letter_Number}`: a number that looks like a letter, such as a Roman numeral.
  - `\p{No}` or `\p{Other_Number}`: a superscript or subscript digit, or a number that is not a digit 0–9 (excluding numbers from ideographic scripts).
- `\p{P}` or `\p{Punctuation}`: any kind of punctuation character.
  - `\p{Pd}` or `\p{Dash_Punctuation}`: any kind of hyphen or dash.
  - `\p{Ps}` or `\p{Open_Punctuation}`: any kind of opening bracket.
  - `\p{Pe}` or `\p{Close_Punctuation}`: any kind of closing bracket.
  - `\p{Pi}` or `\p{Initial_Punctuation}`: any kind of opening quote.
  - `\p{Pf}` or `\p{Final_Punctuation}`: any kind of closing quote.
  - `\p{Pc}` or `\p{Connector_Punctuation}`: a punctuation character such as an underscore that connects words.

    - `\p{Po}` or `\p{Other_Punctuation}`: any kind of punctuation character that is not a dash, bracket, quote or connector.
- `\p{C}` or `\p{Other}`: invisible control characters and unused code points.
    - `\p{Cc}` or `\p{Control}`: an ASCII or Latin-1 control character: 0x00–0x1F and 0x7F–0x9F.
    - `\p{Cf}` or `\p{Format}`: invisible formatting indicator.
    - `\p{Co}` or `\p{Private_Use}`: any code point reserved for private use.
    - `\p{Cs}` or `\p{Surrogate}`: one half of a surrogate pair in UTF-16 encoding.
    - `\p{Cn}` or `\p{Unassigned}`: any code point to which no character has been assigned.

Unicode Scripts

The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Some scripts like `Thai` correspond with a single human language. Other scripts like `Latin` span multiple languages.

Some languages are composed of multiple scripts. There is no Japanese Unicode script. Instead, Unicode offers the `Hiragana`, `Katakana`, `Han`, and `Latin` scripts that Japanese documents are usually composed of.

A special script is the `Common` script. This script contains all sorts of characters that are common to a wide range of scripts. It includes all sorts of punctuation, whitespace and miscellaneous symbols.

All assigned Unicode code points (those matched by `\P{Cn}`) are part of exactly one Unicode script. All unassigned Unicode code points (those matched by `\p{Cn}`) are not part of any Unicode script at all.

The JGsoft engine, Perl, PCRE, PHP, Ruby 1.9, Delphi, and XRegExp can match Unicode scripts. Here's a list:

1. `\p{Common}`
2. `\p{Arabic}`
3. `\p{Armenian}`
4. `\p{Bengali}`
5. `\p{Bopomofo}`
6. `\p{Braille}`
7. `\p{Buhid}`
8. `\p{Canadian_Aboriginal}`
9. `\p{Cherokee}`
10. `\p{Cyrillic}`
11. `\p{Devanagari}`
12. `\p{Ethiopic}`
13. `\p{Georgian}`
14. `\p{Greek}`
15. `\p{Gujarati}`
16. `\p{Gurmukhi}`
17. `\p{Han}`
18. `\p{Hangul}`
19. `\p{Hanunoo}`
20. `\p{Hebrew}`
21. `\p{Hiragana}`
22. `\p{Inherited}`
23. `\p{Kannada}`
24. `\p{Katakana}`

25. `\p{Khmer}`
26. `\p{Lao}`
27. `\p{Latin}`
28. `\p{Limbu}`
29. `\p{Malayalam}`
30. `\p{Mongolian}`
31. `\p{Myanmar}`
32. `\p{Ogham}`
33. `\p{Oriya}`
34. `\p{Runic}`
35. `\p{Sinhala}`
36. `\p{Syriac}`
37. `\p{Tagalog}`
38. `\p{Tagbanwa}`
39. `\p{TaiLe}`
40. `\p{Tamil}`
41. `\p{Telugu}`
42. `\p{Thaana}`
43. `\p{Thai}`
44. `\p{Tibetan}`
45. `\p{Yi}`

Perl and the JGsoft flavor allow you to use `\p{IsLatin}` instead of `\p{Latin}`. The “Is” syntax is useful for distinguishing between scripts and blocks, as explained in the next section. PCRE, PHP, and XRegExp do not support the “Is” prefix.

Java 7 adds support for Unicode scripts. Unlike the other flavors, Java 7 requires the “Is” prefix.

Unicode Blocks

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Each block is used to define characters of a particular script like “Tibetan” or belonging to a particular group like “Braille Patterns”. Most blocks include unassigned code points, reserved for future expansion of the Unicode standard.

Note that Unicode blocks do not correspond 100% with scripts. An essential difference between blocks and scripts is that a block is a single contiguous range of code points, as listed below. Scripts consist of characters taken from all over the Unicode character map. Blocks may include unassigned code points (i.e. code points matched by `\p{Cn}`). Scripts never include unassigned code points. Generally, if you’re not sure whether to use a Unicode script or Unicode block, use the script.

For example, the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, even though both are currency symbols, and the yen symbol is not a Latin character. This is for historical reasons, because the ASCII standard includes the dollar sign, and the ISO-8859 standard includes the yen sign. You should not blindly use any of the blocks listed below based on their names. Instead, look at the ranges of characters they actually match. A tool like RegexBuddy can be very helpful with this. The Unicode property `\p{Sc}` or `\p{Currency_Symbol}` would be a better choice than the Unicode block `\p{InCurrency_Symbols}` when trying to find all currency symbols.

1. `\p{InBasic_Latin}`: U+0000–U+007F
2. `\p{InLatin-1_Supplement}`: U+0080–U+00FF

3. `\p{InLatin_Extended-A}`: U+0100–U+017F
4. `\p{InLatin_Extended-B}`: U+0180–U+024F
5. `\p{InIPA_Extensions}`: U+0250–U+02AF
6. `\p{InSpacing_Modifier_Letters}`: U+02B0–U+02FF
7. `\p{InCombining_Diacritical_Marks}`: U+0300–U+036F
8. `\p{InGreek_and_Coptic}`: U+0370–U+03FF
9. `\p{InCyrillic}`: U+0400–U+04FF
10. `\p{InCyrillic_Supplementary}`: U+0500–U+052F
11. `\p{InArmenian}`: U+0530–U+058F
12. `\p{InHebrew}`: U+0590–U+05FF
13. `\p{InArabic}`: U+0600–U+06FF
14. `\p{InSyriac}`: U+0700–U+074F
15. `\p{InThaana}`: U+0780–U+07BF
16. `\p{InDevanagari}`: U+0900–U+097F
17. `\p{InBengali}`: U+0980–U+09FF
18. `\p{InGurmukhi}`: U+0A00–U+0A7F
19. `\p{InGujarati}`: U+0A80–U+0AFF
20. `\p{InOriya}`: U+0B00–U+0B7F
21. `\p{InTamil}`: U+0B80–U+0BFF
22. `\p{InTelugu}`: U+0C00–U+0C7F
23. `\p{InKannada}`: U+0C80–U+0CFF
24. `\p{InMalayalam}`: U+0D00–U+0D7F
25. `\p{InSinhala}`: U+0D80–U+0DFF
26. `\p{InThai}`: U+0E00–U+0E7F
27. `\p{InLao}`: U+0E80–U+0EFF
28. `\p{InTibetan}`: U+0F00–U+0FFF
29. `\p{InMyanmar}`: U+1000–U+109F
30. `\p{InGeorgian}`: U+10A0–U+10FF
31. `\p{InHangul_Jamo}`: U+1100–U+11FF
32. `\p{InEthiopic}`: U+1200–U+137F
33. `\p{InCherokee}`: U+13A0–U+13FF
34. `\p{InUnified_Canadian_Aboriginal_Syllabics}`: U+1400–U+167F
35. `\p{InOgham}`: U+1680–U+169F
36. `\p{InRunic}`: U+16A0–U+16FF
37. `\p{InTagalog}`: U+1700–U+171F
38. `\p{InHanunoo}`: U+1720–U+173F
39. `\p{InBuhid}`: U+1740–U+175F
40. `\p{InTagbanwa}`: U+1760–U+177F
41. `\p{InKhmer}`: U+1780–U+17FF
42. `\p{InMongolian}`: U+1800–U+18AF
43. `\p{InLimbu}`: U+1900–U+194F
44. `\p{InTai_Le}`: U+1950–U+197F
45. `\p{InKhmer_Symbols}`: U+19E0–U+19FF
46. `\p{InPhonetic_Extensions}`: U+1D00–U+1D7F
47. `\p{InLatin_Extended_Additional}`: U+1E00–U+1EFF
48. `\p{InGreek_Extended}`: U+1F00–U+1FFF
49. `\p{InGeneral_Punctuation}`: U+2000–U+206F
50. `\p{InSuperscripts_and_Subscripts}`: U+2070–U+209F
51. `\p{InCurrency_Symbols}`: U+20A0–U+20CF

52. `\p{InCombining_Diacritical_Marks_for_Symbols}`: U+20D0–U+20FF
53. `\p{InLetterlike_Symbols}`: U+2100–U+214F
54. `\p{InNumber_Forms}`: U+2150–U+218F
55. `\p{InArrows}`: U+2190–U+21FF
56. `\p{InMathematical_Operators}`: U+2200–U+22FF
57. `\p{InMiscellaneous_Technical}`: U+2300–U+23FF
58. `\p{InControl_Pictures}`: U+2400–U+243F
59. `\p{InOptical_Character_Recognition}`: U+2440–U+245F
60. `\p{InEnclosed_Alphanumerics}`: U+2460–U+24FF
61. `\p{InBox_Drawing}`: U+2500–U+257F
62. `\p{InBlock_Elements}`: U+2580–U+259F
63. `\p{InGeometric_Shapes}`: U+25A0–U+25FF
64. `\p{InMiscellaneous_Symbols}`: U+2600–U+26FF
65. `\p{InDingbats}`: U+2700–U+27BF
66. `\p{InMiscellaneous_Mathematical_Symbols-A}`: U+27C0–U+27EF
67. `\p{InSupplemental_Arrows-A}`: U+27F0–U+27FF
68. `\p{InBraille_Patterns}`: U+2800–U+28FF
69. `\p{InSupplemental_Arrows-B}`: U+2900–U+297F
70. `\p{InMiscellaneous_Mathematical_Symbols-B}`: U+2980–U+29FF
71. `\p{InSupplemental_Mathematical_Operators}`: U+2A00–U+2AFF
72. `\p{InMiscellaneous_Symbols_and_Arrows}`: U+2B00–U+2BFF
73. `\p{InCJK_Radicals_Supplement}`: U+2E80–U+2EFF
74. `\p{InKangxi_Radicals}`: U+2F00–U+2FDF
75. `\p{InIdeographic_Description_Characters}`: U+2FF0–U+2FFF
76. `\p{InCJK_Symbols_and_Punctuation}`: U+3000–U+303F
77. `\p{InHiragana}`: U+3040–U+309F
78. `\p{InKatakana}`: U+30A0–U+30FF
79. `\p{InBopomofo}`: U+3100–U+312F
80. `\p{InHangul_Compatibility_Jamo}`: U+3130–U+318F
81. `\p{InKanbun}`: U+3190–U+319F
82. `\p{InBopomofo_Extended}`: U+31A0–U+31BF
83. `\p{InKatakana_Phonetic_Extensions}`: U+31F0–U+31FF
84. `\p{InEnclosed_CJK_Letters_and_Months}`: U+3200–U+32FF
85. `\p{InCJK_Compatibility}`: U+3300–U+33FF
86. `\p{InCJK_Unified_Ideographs_Extension_A}`: U+3400–U+4DBF
87. `\p{InYijing_Hexagram_Symbols}`: U+4DC0–U+4DFF
88. `\p{InCJK_Unified_Ideographs}`: U+4E00–U+9FFF
89. `\p{InYi_Syllables}`: U+A000–U+A48F
90. `\p{InYi_Radicals}`: U+A490–U+A4CF
91. `\p{InHangul_Syllables}`: U+AC00–U+D7AF
92. `\p{InHigh_Surrogates}`: U+D800–U+DB7F
93. `\p{InHigh_Private_Use_Surrogates}`: U+DB80–U+DBFF
94. `\p{InLow_Surrogates}`: U+DC00–U+DFFF
95. `\p{InPrivate_Use_Area}`: U+E000–U+F8FF
96. `\p{InCJK_Compatibility_Ideographs}`: U+F900–U+FAFF
97. `\p{InAlphabetic_Presentation_Forms}`: U+FB00–U+FB4F
98. `\p{InArabic_Presentation_Forms-A}`: U+FB50–U+FDFF
99. `\p{InVariation_Selectors}`: U+FE00–U+FE0F
100. `\p{InCombining_Half_Marks}`: U+FE20–U+FE2F

101. `\p{InCJK_Compatibility_Forms}`: U+FE30–U+FE4F
102. `\p{InSmall_Form_Variants}`: U+FE50–U+FE6F
103. `\p{InArabic_Presentation_Forms-B}`: U+FE70–U+FEFF
104. `\p{InHalfwidth_and_Fullwidth_Forms}`: U+FF00–U+FFEF
105. `\p{InSpecials}`: U+FFF0–U+FFFF

Not all Unicode regex engines use the same syntax to match Unicode blocks. Java, Ruby 2.0, and XRegExp use the `\p{InBlock}` syntax as listed above. .NET and XML use `\p{IsBlock}` instead. Perl and the JGsoft flavor support both notations. I recommend you use the "In" notation if your regex engine supports it. "In" can only be used for Unicode blocks, while "Is" can also be used for Unicode properties and scripts, depending on the regular expression flavor you're using. By using "In", it's obvious you're matching a block and not a similarly named property or script.

In .NET and XML, you must omit the underscores but keep the hyphens in the block names. E.g. Use `\p{IsLatinExtended-A}` instead of `\p{InLatin_Extended-A}`. In Java, you must omit the hyphens. .NET and XML also compare the names case sensitively, while Perl, Ruby, and the JGsoft flavor compare them case insensitively. Java 4 is case sensitive. Java 5 and later are case sensitive for the "Is" prefix but not for the block names themselves.

The actual names of the blocks are the same in all regular expression engines. The block names are defined in the Unicode standard. PCRE and PHP do not support Unicode blocks, even though they support Unicode scripts.

Do You Need To Worry About Different Encodings?

While you should always keep in mind the pitfalls created by the different ways in which accented characters can be encoded, you don't always have to worry about them. If you know that your input string and your regex use the same style, then you don't have to worry about it at all. This process is called Unicode *normalization*. All programming languages with native Unicode support, such as Java, C# and VB.NET, have library routines for normalizing strings. If you normalize both the subject and regex before attempting the match, there won't be any inconsistencies.

If you are using Java, you can pass the CANON_EQ flag as the second parameter to Pattern.compile(). This tells the Java regex engine to consider *canonically equivalent* characters as identical. The regex à encoded as U+00E0 matches à encoded as U+0061 U+0300, and vice versa. None of the other regex engines currently support canonical equivalence while matching.

If you type the à key on the keyboard, all word processors that I know of will insert the code point U+00E0 into the file. So if you're working with text that you typed in yourself, any regex that you type in yourself will match in the same way.

Finally, if you're using PowerGREP to search through text files encoded using a traditional Windows (often called "ANSI") or ISO-8859 code page, PowerGREP always uses the one-on-one substitution. Since all the Windows or ISO-8859 code pages encode accented characters as a single code point, nearly all software uses a single Unicode code point for each character when converting the file to Unicode.

Make a Donation

Did this website just save you a trip to the bookstore? Please make a donation to support this site, and you'll get a **lifetime of advertisement-free access** to this site!