Examples

- Regular Expressions Examples
- Numeric Ranges
- Floating Point Numbers
- Email Addresses
- IP Addresses
- Valid Dates
- Numeric Dates to Text
- Credit Card Numbers
- Matching Complete Lines
- Deleting Duplicate Lines
- Programming
- Two Near Words

Pitfalls

- Catastrophic Backtracking
- Too Many Repetitions
- Denial of Service
- Making Everything Optional
- Repeated Capturing Group
- Mixing Unicode & 8-bit

More on This Site

- Introduction
- Regular Expressions Quick Start
- Regular Expressions Tutorial
- Replacement Strings Tutorial
- Applications and Languages
- Regular Expressions Examples
- Regular Expressions Reference
- Replacement Strings Reference
- Book Reviews
- Printable PDF
- About This Site
- RSS Feed & Blog

How to Find or Validate an Email Address

The regular expression I receive the most feedback, not to mention "bug" reports on, is the one you'll find right on this site's home page: `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b`. This regular expression, I claim, matches any email address. Most of the feedback I get refutes that claim by showing one email address that this regex doesn't match. Usually, the "bug" report also includes a suggestion to make the regex "perfect".

As I explain below, my claim only holds true when one accepts my definition of what a valid email address really is, and what it's not. If you want to use a different definition, you'll have to adapt the regex. Matching a valid email address is a perfect example showing that (1) before writing a regex, you have to know exactly what you're trying to match, and what not; and (2) there's often a trade-off between what's exact, and what's practical.

The virtue of my regular expression above is that it matches 99% of the email addresses in use today. All the email addresses it matches can be handled by 99% of all email software out there. If you're looking for a quick solution, you only need to read the next paragraph. If you want to know all the trade-offs and get plenty of alternatives to choose from, read on.

If you want to use the regular expression above, there are two things you need to understand. First, long regexes make it difficult to nicely format paragraphs. So I didn't include `a-z` in any of the three character classes. This regex is intended to be used with your regex engine's "case insensitive" option turned on. (You'd be surprised how many "bug" reports I get about that.) Second, the above regex is delimited with word boundaries, which makes it suitable for extracting email addresses from files or larger blocks of text. If you want to check whether the user typed in a valid email address, replace the word boundaries with start-of-string and end-of-string anchors, like this: `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$`.

The previous paragraph also applies to all of the following examples. You may need to change word boundaries into start/end-of-string anchors, or vice versa. And you have to turn on the case insensitive matching option.

Trade-Offs in Validating Email Addresses

Before ICANN made it possible for any well-funded company to create their own top-level domains, the longest top-level domains were the rarely used `.museum` and `.travel` which are 6 letters long. The most common top-level domains were 2 letters long for country-specific domains, and 3 or 4 letters long for general-purpose domains like `.com` and `.info`. A lot of regexes for validating email addresses you'll find in various regex tutorials and references still assume the top-level domain to be fairly short. Older editions of this regex tutorial mentioned `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b` as the regex for email addresses in its introduction. There's only one little difference between this regex and the one at the top of this page. The `4` at the end of the regex restricts the top-level domain to 4 characters. If you use this regex with anchors to validate the email address entered on your order form, `fabio@disapproved.solutions` has to do his shopping elsewhere. Yes, the `.solutions` TLD exists and when I write this, `disaproved.solutions` can be yours for $16.88 per year.

If you want to be more strict than `[A-Z]{2,}` for the top-level domain, `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,63}$` is as far as you can practically go. Each part of a domain name can be no longer than 63 characters. There are no single-digit top-level domains and none contain digits. It doesn't look like ICANN will approve such domains either.

Email addresses can be on servers on a subdomain as in `john@server.department.company.com`. All of the above regexes match this email address, because I included a dot in the character class after the @ symbol. But the above regexes also match `john@aol...com` which is not valid due to the consecutive dots. You can exclude such matches by replacing `[A-Z0-9.-]+\.` with `(?:[A-Z0-9-]+\.)+` in any of the above regexes. I removed the dot from the character class and instead repeated the character class and the following literal dot. E.g. `^[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,}$` matches `john@server.department.company.com` but not `john@aol...com`.

If you want to avoid your system choking on arbitrarily large input, you can replace the infinite quantifiers with finite ones. `^[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$` takes into account that the local part (before the @) is limited to 64 characters and that each part of the domain name is limited to 63 characters. There's no direct limit on the number of subdomains. But the maximum length of an email address that can be handled by SMTP is 254 characters. So with a single-character local part, a two-letter top-level domain and single-character sub-domains, 125 is the maximum number of sub-domains.

The previous regex does not actually limit email addresses to 254 characters. If each part is at its maximum length, the regex can match strings up to 8129 characters in length. You can reduce that by lowering the number of allowed sub-domains from 125 to something more realistic like 8. I've never seen an email address with more than 4 subdomains. If you want to enforce the 254 character limit, the best solution is to check the length of the input string before you even use a regex. Though this requires a few lines of procedural code, checking the length of a string is near-instantaneous. If you can only use regexes, `^[A-Z0-9@._%+-]{6,254}$` can be used as a first pass to make sure the string doesn't contain invalid characters and isn't too short or too long. If you need to do everything with one regex, you'll need a regex flavor that supports lookahead. The regular expression `^(?=[A-Z0-9@._%+-]{6,254}$)[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$` uses a lookahead to first check that the string doesn't contain invalid characters and isn't too short or too long. When the lookahead succeeds, the remainder of the regex makes a second pass over the string to check for proper placement of the @ sign and the dots.

All of these regexes allow the characters `._%+-` anywhere in the local part. You can force the local part to begin with a letter by using `^[A-Z0-9][A-Z0-9._%+-]{0,63}` instead of `^[A-Z0-9._%+-]{1,64}` for the local part: `^[A-Z0-9][A-Z0-9._%+-]{0,63}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$`. When using lookahead to check the overall length of the address, the first character can be checked in the lookahead. We don't need to repeat the initial character check when checking the length of the local part. This regex is too long to fit the width of the page, so let's turn on free-spacing mode:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}$)
[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$
```

Domain names can contain hyphens. But they cannot begin or end with a hyphen. `[A-Z0-9](?:[A-Z0-9-]{0,62}[A-Z0-9])?` matches a domain name between 1 and 63 characters long that starts and ends with a letter or digit. The non-capturing group makes the middle of the domain and the final letter or digit optional as a whole to ensure that we allow single-character domains while at the same time ensuring that domains with two or more characters do not end with a hyphen. The overall regex starts to get quite complicated:

```
^[A-Z0-9][A-Z0-9._%+-]{0,63}@
(?:[A-Z0-9](?:[A-Z0-9-]{0,62}[A-Z0-9])?\.){1,8}[A-Z]{2,63}$
```

Domain names cannot contain consecutive hyphens. `[A-Z0-9]+(?:-[A-Z0-9]+)*` matches a domain name that starts and ends with a letter or digit and that contains any number of non-consecutive hyphens. This is the most efficient way. This regex does not do any backtracking to match a valid domain name. It matches all letters and digits at the start of the domain name. If there are no hyphens, the optional group that follows fails immediately. If there are hyphens, the group matches each hyphen followed by all letters and digits up to the next hyphen or the end of the domain name. We can't enforce the maximum length when hyphens must be paired with a letter or digit, but letters and digits can stand on their own. But we can use the lookahead technique that we used to enforce the overall length of the email address to enforce the length of the domain name while disallowing consecutive hyphens: `(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*`. Notice that the lookahead also checks for the dot that must appear after the domain name when it is fully qualified in an email address. This is important. Without checking for the dot, the lookahead would accept longer domain names. Since the lookahead does not consume the text it matches, the dot is not included in the overall match of this regex. When we put this regex into the overall regex for email addresses, the dot will be matched as it was in the previous regexes:

```
^[A-Z0-9][A-Z0-9._%+-]{0,63}@
(?:(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*\.){1,8}[A-Z]{2,63}$
```

If we include the lookahead to check the overall length, our regex makes two passes over the local part, and three passes over the domain names to validate everything:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}$)[A-Z0-9._%+-]{1,64}@
(?:(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*\.){1,8}[A-Z]{2,63}$
```

On a modern PC or server this regex will perform just fine when validating a single 254-character email address. Rejecting longer input would even be faster because the regex will fail when the lookahead fails during first pass. But I wouldn't recommend using a regex as complex as this to search for email addresses through a large archive of documents or correspondence. You're better off using the simple regex at the top of this page to quickly gather everything that looks like an email address. Deduplicate the results and then use a stricter regex if you want to further

filter out invalid addresses.

And speaking of backtracking, none of the regexes on this page do any backtracking to match valid email addresses. But particularly the latter ones may do a fair bit of backtracking on something that's not quite a valid email address. If your regex flavor supports possessive quantifiers, you can eliminate all backtracking by making all quantifiers possessive. Because no backtracking is needed to find matches, doing this does not change what is matched by these regexes. It only allows them to fail faster when the input is not a valid email address. The simplest regex that correctly handles subdomains then becomes `^[A-Z0-9._%+-]++@(?:[A-Z0-9-]++\.)++[A-Z]{2,}+$` with an extra + after each quantifier. We can do the same with our most complex regex:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}+$)[A-Z0-9._%+-]{1,64}+@
(?:(?=[A-Z0-9-]{1,63}+\.)[A-Z0-9]++(?:-[A-Z0-9]++)*+\.){1,8}+[A-Z]{2,63}+$
```

An important trade-off in all these regexes is that they only allow English letters, digits, and the most commonly used special symbols. The main reason is that I don't trust all my email software to be able to handle much else. Even though `John.O'Hara@theoharas.com` is a syntactically valid email address, there's a risk that some software will misinterpret the apostrophe as a delimiting quote. Blindly inserting this email address into an SQL query, for example, will at best cause it to fail when strings are delimited with single quotes and at worst open your site up to SQL injection attacks.

And of course, it's been many years already that domain names can include non-English characters. But most software still sticks to the 37 characters Western programmers are used to. Supporting internationalized domains opens up a whole can of worms of how the non-ASCII characters should be encoded. So if you use any of the regexes on this page, anyone with an @ที่เอชนิค.ไทย address will be out of luck. But perhaps it is telling that http://ที่เอชนิค.ไทย simply redirects to http://thnic.co.th even though they're in the business of selling .ไทย domains.

The conclusion is that to decide which regular expression to use, whether you're trying to match an email address or something else that's vaguely defined, you need to start with considering all the trade-offs. How bad is it to match something that's not valid? How bad is it not to match something that is valid? How complex can your regular expression be? How expensive would it be if you had to change the regular expression later because it turned out to be too broad or too narrow? Different answers to these questions will require a different regular expression as the solution. My email regex does what I want, but it may not do what you want.

Regexes Don't Send Email

Don't go overboard in trying to eliminate invalid email addresses with your regular expression. The reason is that you don't really know whether an address is valid until you try to send an email to it. And even that might not be enough. Even if the email arrives in a mailbox, that doesn't mean somebody still reads that mailbox. If you really need to be sure an email address is valid, you'll need to send an email to it that contains a code or link for the recipient to perform a second authentication step. And if you're doing that, then there is little point in using a regex that may reject valid email addresses.

The same principle applies in many situations. When trying to match a valid date, it's often easier to use a bit of arithmetic to check for leap years, rather than trying to do it in a regex. Use a regular expression to find potential matches or check if the input uses the proper syntax, and do the actual validation on the potential matches returned by the regular expression. Regular expressions are a powerful tool, but they're far from a panacea.

The Official Standard: RFC 5322

Maybe you're wondering why there's no "official" fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof.

The official standard is known as RFC 5322. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't—read on) implement it with the following regular expression. RFC 5322 leaves the domain name part open to implementation-specific choices that won't work on the Internet today. The regex implements the "preferred" syntax from RFC 1035 which is one of the recommendations in RFC 5322:

```
\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
  |  "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
        |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
        (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
           (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
           |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
    \])\z
```

This regex has two parts: the part before the @, and the part after the @. There are two alternatives for the part before the @. The first alternative allows it to consist of a series of letters, digits and certain symbols, including one or more dots. However, dots may not appear consecutively or at the start or end of the email address. The other alternative requires the part before the @ to be enclosed in double quotes, allowing any string of ASCII characters between the quotes. Whitespace characters, double quotes and backslashes must be escaped with backslashes.

The part after the @ also has two alternatives. It can either be a fully qualified domain name (e.g. regular-expressions.info), or it can be a literal Internet address between square brackets. The literal Internet address can either be an IP address, or a domain-specific routing address.

The reason you shouldn't use this regex is that it is overly broad. Your application may not be able to handle all email addresses this regex allows. Domain-specific routing addresses can contain non-printable ASCII control characters, which can cause trouble if your application needs to display addresses. Not all applications support the syntax for the local part using double quotes or square brackets. In fact, RFC 5322 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 5322 if we omit IP addresses, domain-specific addresses, the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
\A[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@
(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
```

Neither of these regexes enforce length limits on the overall email address or the local part or the domain names. RFC 5322 does not specify any length limitations. Those stem from limitations in other protocols like the SMTP protocol for actually sending email. RFC 1035 does state that domains must be 63 characters or less, but does not include that in its syntax specification. The reason is that a true regular language cannot enforce a length limit and disallow consecutive hyphens at the same time. But modern regex flavors aren't truly regular, so we can add length limit checks using lookahead like we did before:

```
\A(?=[a-z0-9@.!#$%&'*+/=?^_`{|}~-]{6,254}\z)
  (?=[a-z0-9.!#$%&'*+/=?^_`{|}~-]{1,64}@)
  [a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
@ (?:(?=[a-z0-9-]{1,63}\.)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
  (?=[a-z0-9-]{1,63}\z)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
```

So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

Page URL: https://www.regular-expressions.info/email.html
Page last updated: 2 September 2021
Site last updated: 02 December 2022