computerhope.com

How do I Extract Specific Portions of a Text File Using Python?

Computer Hope

23-29 minutes

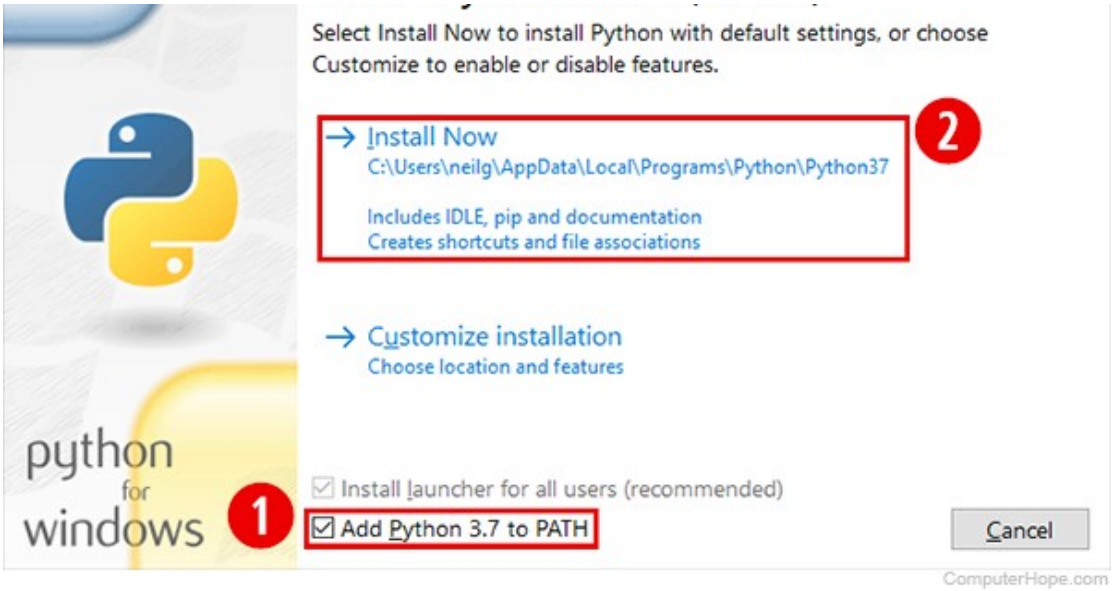
Updated: 05/04/2019 by

Extracting text from a file is a common task in scripting and programming, and Python makes it easy. In this guide, we'll discuss some simple ways to extract text from a file using the Python 3 programming language.

Make sure you're using Python 3

In this guide, we'll be using Python version 3. Most systems come pre-installed with Python 2.7. While Python 2.7 is used in most legacy code, Python 3 is the present and future of the Python language. Unless you have a specific reason to write or support legacy Python code, we recommend working in Python 3.

For Microsoft Windows, Python 3 can be downloaded from https://www.python.org. When installing, make sure the "Install launcher for all users" and "Add Python to PATH" options are both checked, as shown in the image below.



On Linux, you can install Python 3 with your package manager. For instance, on Debian or Ubuntu, you can install it with the following command:

```
sudo apt-get update && sudo apt-get install python3
```

For macOS, the Python 3 installer can be downloaded from python.org, as linked above. If you are using the Homebrew package manager, it can also be installed by opening a terminal window (**Applications → Utilities**), and running this command:

```
brew install python3
```

Running Python

On Linux and macOS, the command to run the Python 3 interpreter is **python3**. On Windows, if you installed the launcher, the command is **py**. The commands on this page use **python3**; if you're on Windows, substitute **py** for **python3** in all commands.

Running Python with no options will start the interactive interpreter. For more information about using the interpreter, see Python overview: using the Python interpreter. If you accidentally enter the interpreter, you can exit it using the command **exit()** or **quit()**.

Running Python with a file name will interpret that python program. For instance:

```
python3 program.py
```

...runs the program contained in the file **program.py**.

Okay, how can we use Python to extract text from a text file?

Reading data from a text file

First, let's read a text file. Let's say we're working with a file named **lorem.txt**, which contains a few lines of Latin:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Mauris nec maximus purus. Maecenas sit amet pretium tellus.
Quisque at dignissim lacus.
```

Note

In all the examples that follow, we work with text contained in this file. Feel free to copy and paste the latin text above into a text file, and save it as **lorem.txt**, so that you can run the example code using this file as input.

A Python program can read a text file using the built-in **open()** function. For example, below is a Python 3 program that opens **lorem.txt** for reading in text mode, reads the contents into a string variable named **contents**, closes the file, and then prints the data.

```
myfile = open("lorem.txt", "rt") # open lorem.txt for reading
text
contents = myfile.read()         # read the entire file into a
string
myfile.close()                   # close the file
print(contents)                  # print contents
```

Here, **myfile** is the name we give to our file object.

The "**rt**" parameter in the **open()** function means "we're opening this file to **r**ead **t**ext data"

The hash mark ("#") means that everything on the rest of that line is a comment, and it is ignored by the Python interpreter.

If you save this program in a file called **read.py**, you can run it with the following command.

```
python3 read.py
```

The command above outputs the contents of **lorem.txt**:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Mauris nec maximus purus. Maecenas sit amet pretium tellus.
Quisque at dignissim lacus.
```

Using "with open"

It's important to close your open files as soon as possible: open the file, perform your operation, and close it. Don't leave it open for extended periods of time.

When you're working with files, it's good practice to use the with open...as compound statement. It's the cleanest way to open a file, operate on it, and close the file, all in one easy-to-read block of code. The file is automatically closed when the code block completes.

Using **with open**...**as**, we can rewrite our program to look like this:

```
with open ('lorem.txt', 'rt') as myfile:  # Open lorem.txt for
reading text
    contents = myfile.read()              # Read the entire file
into a string
print(contents)                           # Print the string
```

Note

Indentation is important in Python. Python programs use white space at the beginning of a line to define scope, such as a block of code. It's recommended that you use four spaces per level of indentation, and that you use spaces rather than tabs. In the following examples, make sure your code is indented exactly as it's presented here.

Example

Save the program as **read.py** and execute it:

```
python3 read.py
```

Output

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Mauris nec maximus purus. Maecenas sit amet pretium tellus.
Quisque at dignissim lacus.
```

Reading text files line-by-line

In the examples so far, we've been reading in the whole file at once. Reading a full file is no big deal with small files, but generally speaking, it's not a great idea. For one thing, if your file is bigger than the amount of available memory, you'll encounter an error.

In almost every case, it's a better idea to read a text file one line at a time.

In Python, the file object is an iterator. An iterator is a type of Python object which behaves in certain ways when operated on repeatedly. For instance, you can use a for loop to operate on a file object repeatedly, and each time the same operation is performed, you'll receive a different, or "next," result.

Example

For text files, the file object iterates one line of text at a time. It considers one line of text a "unit" of data, so we can use a for...in loop statement to iterate on the data one line at a time:

```
with open ('lorem.txt', 'rt') as myfile:  # Open file lorem.txt
for reading text
    for myline in myfile:                 # For each line, read
it to a string
        print(myline)                     # print that string,
```

```
repeat
```

Output

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Mauris nec maximus purus. Maecenas sit amet pretium tellus.

Quisque at dignissim lacus.
```

Notice that we're getting an extra line break ("newline") after every line. That's because two newlines are being printed. The first one is the newline at the end of every line of our text file. The second newline happens because, by default, print() adds a linebreak of its own at the end of whatever you've asked it to print.

Let's store our lines of text in a variable — specifically, a *list* variable — so we can look at it more closely.

Storing text data in a list variable

In Python, lists are similar to, but not the same as, an array in C or Java. A Python list contains indexed data, of varying lengths and types.

Example

```
mylines = []                             # Declare an empty list
named mylines.
with open ('lorem.txt', 'rt') as myfile: # Open lorem.txt for
reading text data.
    for myline in myfile:                # For each line, stored
as myline,
        mylines.append(myline)           # add its contents to
mylines.
print(mylines)                           # Print the list.
```

The output of this program is a little different. Instead of printing the *contents* of the list, this program prints our list *object*, which looks like this:

Output

```
['Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc
fringilla arcu congue metus aliquam mollis.\n', 'Mauris nec
maximus purus. Maecenas sit amet pretium tellus. Praesent sed
rhoncus eo. Duis id commodo orci.\n', 'Quisque at dignissim
lacus.\n']
```

Here, we see the raw contents of the list. In its raw object form, a list is represented as a

comma-delimited list. Here, each element is represented as a string, and each newline is represented as its escape character sequence, **\n**.

Much like an array in C or Java, we can access the elements of a list by specifying an index number after the variable name, in brackets. Index numbers start at zero — other words, the *n*th element of a list has the numeric index *n*-1.

Note

If you're wondering why the index numbers start at zero instead of one, you're not alone. Computer scientists have debated the usefulness of zero-based numbering systems in the past. In 1982, Edsger Dijkstra gave his opinion on the subject, explaining why zero-based numbering is the best way to index data in computer science. You can read the memo yourself — he makes a compelling argument.

Example

We can print the first element of **lines** by specifying index number **0**, contained in brackets after the name of the list:

```
print(mylines[0])
```

Output

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc
fringilla arcu congue metus aliquam mollis.
```

Example

Or the third line, by specifying index number 2:

```
print(mylines[2])
```

Output

```
Quisque at dignissim lacus.
```

But if we try to access an index for which there is no value, we get an error:

Example

```
print(mylines[3])
```

Output

```
Traceback (most recent call last):
File <filename>, line <linenum>, in <module>
print(mylines[3])
IndexError: list index out of range
```

Example

A list object is an iterator, so to print every element of the list, we can iterate over it with **for**...**in**:

```
mylines = []                                # Declare an empty list
with open ('lorem.txt', 'rt') as myfile:  # Open lorem.txt for
reading text.
    for line in myfile:                     # For each line of
text,
        mylines.append(line)                # add that line to the
list.
    for element in mylines:                 # For each element in
the list,
        print(element)                      # print it.
```

Output

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Mauris nec maximus purus. Maecenas sit amet pretium tellus.

Quisque at dignissim lacus.
```

But we're still getting extra newlines. Each line of our text file ends in a newline character ('**\n**'), which is being printed. Also, after printing each line, **print()** adds a newline of its own, unless you tell it to do otherwise.

We can change this default behavior by specifying an **end** parameter in our **print()** call:

```
print(element, end='')
```

By setting **end** to an empty string (represented as two single quotes, with no space between), we tell **print()** to print *nothing* at the end of a line, instead of a newline character.

Example

Our revised program looks like this:

```
mylines = []                                # Declare an empty list
with open ('lorem.txt', 'rt') as myfile:  # Open file lorem.txt
for reading text
    for line in myfile:                     # For each line of
text,
        mylines.append(line)                # add that line to the
list.
    for element in mylines:                 # For each element in
```

```
the list,
        print(element, end='')            # print it, without
extra newlines.
```

Output

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Mauris nec maximus purus. Maecenas sit amet pretium tellus.
Quisque at dignissim lacus.
```

The newlines you see here are actually in the file; they're a special character ('**\n**') at the end of each line. We want to get rid of these, so we don't have to worry about them while we process the file.

How to strip newlines

To remove the newlines completely, we can strip them. To **strip** a string is to remove one or more characters, usually whitespace, from either the beginning or end of the string.

Tip

This process is sometimes also called "trimming."

Python 3 string objects have a method called **rstrip()**, which strips characters from the right side of a string. The English language reads left-to-right, so stripping from the right side removes characters from the end.

If the variable is named **mystring**, we can strip its right side with **mystring.rstrip(*chars*)**, where *chars* is a string of characters to strip, if they are on the right side of the string. For example, **"123abc".rstrip("bc")** returns **123a**.

Tip

When you represent a string in your program with its literal contents, it's called a string literal. In Python (as in most programming languages), string literals are always *quoted* — enclosed on either side by single (') or double (") quotes. In Python, single and double quotes are equivalent; you can use one or the other, as long as they match on both ends of the string. It's traditional to represent a human-readable string (such as **Hello**) in double-quotes (**"Hello"**). If you're representing a single character (such as **b**), or a single special character such as the newline character ( ), it's traditional to use single quotes (**'b'**, **' '**). For more information about how to use strings in Python, you can read the documentation of strings in Python.

The statement ***string*.rstrip('\n')** will strip a newline character from the right side of *string*. The following version of our program strips the newlines when each line is read from the text file:

```
mylines = []                                # Declare an empty
list.
with open ('lorem.txt', 'rt') as myfile:    # Open lorem.txt for
```

```
reading text.
    for myline in myfile:                   # For each line in
the file,
        mylines.append(myline.rstrip('\n')) # strip newline and
add to list.
for element in mylines:                     # For each element in
the list,
    print(element)                          # print it.
```

The text is now stored in a list variable, so individual lines can be accessed by index number. Newlines have been stripped, so we don't have to worry about them. We can always put them back later if we reconstruct the file and write it to disk.

Now, let's search the lines in the list for a specific substring.

Searching text for a substring

Let's say we want to locate every occurrence of a certain phrase, or even a single letter. For instance, maybe we need to know where every "e" is. We can accomplish this using the string's find() method.

The list stores each line of our text as a string object. All string objects have a method, **find()**, which locates the first occurrence of a substrings in the string.

Let's use the **find()** method to search for the letter "e" in the first line of our text file, which is stored in the list **mylines**. The first element of **mylines** is a string object containing the first line of the text file. This string object has a **find()** method.

In the parentheses of **find()**, we specify parameters. The first and only required parameter is the string to search for, **"e"**. The statement **mylines[0].find("e")** tells the interpreter to start at the beginning of the string and search forward, one character at a time, until it finds the letter "e." When it finds one, it stops searching, and returns the index number where that "e" is located. If it reaches the end of the string, it returns **-1** to indicate nothing was found.

Example

```
print(mylines[0].find("e"))
```

Output

```
3
```

The return value "3" tells us that the letter "e" is the fourth character, the "e" in "Lorem". (Remember, the index is zero-based: index 0 is the first character, 1 is the second, etc.)

The **find()** method takes two optional, additional parameters: a *start* index and a *stop* index, indicating where in the string the search should begin and end. For instance, ***string*.find("abc", 10, 20)** will search for the substring "abc", but only from the 11th to

the 21st character. If *stop* is not specified, **find()** will start at index *start*, and stop at the end of the string.

Example

For instance, the following statement searchs for "e" in **mylines[0]**, beginning at the fifth character.

```
print(mylines[0].find("e", 4))
```

Output

```
24
```

In other words, starting at the 5th character in line[0], the first "e" is located at index 24 (the "e" in "nec").

Example

To start searching at index 10, and stop at index 30:

```
print(mylines[1].find("e", 10, 30))
```

Output

```
28
```

(The first "e" in "Maecenas").

Example

If **find()** doesn't locate the substring in the search range, it will return the number **-1**, indicating failure:

```
print(mylines[0].find("e", 25, 30))
```

Output

```
-1
```

There were no "e" occurrences between indices 25 and 30.

Finding all occurrences of a substring

But what if we want to locate *every* occurrence of a substring, not just the first one we encounter? We can iterate over the string, starting from the index of the previous match.

In this example, we'll use a while loop to repeatedly **find** the letter "e". When an occurrence is found, we call **find** again, starting from a new location in the string. Specifically, the location of the last occurrence, plus the length of the string (so we can move forward past the last one). When **find** returns **-1**, or the start index exceeds the

length of the string, we stop.

Example

```
# Build mylines as shown above

mylines = []                                # Declare an empty
list.
with open ('lorem.txt', 'rt') as myfile:    # Open lorem.txt for
reading text.
    for myline in myfile:                   # For each line in
the file,
        mylines.append(myline.rstrip('\n')) # strip newline and
add to list.

# Locate and print all occurences of letter "e"

index = 0                     # current index
prev = 0                      # previous index
str = mylines[0]              # string to be searched (first
element of mylines)
substr = "e"                  # substring to search for
while index < len(str):       # While index has not exceeded
string length,
  index = str.find(substr, index)  # set index to first
occurrence of "e"
  if index == -1:             # If nothing was found,
    break                     # exit the while loop.
  print(" " * (index - prev) + "e", end='')  # print location of
this "e"
  prev = index + len(substr)  # set previous to index + 1
  index += len(substr)        # increment the index by the length
of substr.
                              # (Repeat while loop until index >=
len(str))
print('\n' + str);            # Print the original string under
the e's
```

Output

```
   e                    e       e  e               e
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

                          e  e
Nunc fringilla arcu congue metus aliquam mollis.
```

Incorporating regular expressions

For complex searches, you should use regular expressions.

The Python regular expressions module is called **re**. To use it in your program, **import** the module before you use it:

```
import re
```

The **re** module implements regular expressions by compiling a search pattern into a pattern object. Methods of this object can then be used to perform match operations.

For example, let's say you want to search for any word in your document which starts with the letter **d** and ends in the letter **r**. We can accomplish this using the regular expression "**\bd\w*r\b**". What does this mean?

character sequence	meaning
\b	A word boundary matches an empty string (anything, including nothing at all), but only if it appears before or after a non-word character. "Word characters" are the digits 0 through 9, the lowercase and uppercase letters, or an underscore ("_").
d	Lowercase letter **d**.
\w*	**\w** represents any word character, and * is a quantifier meaning "zero or more of the previous character." So **\w*** will match zero or more word characters.
r	Lowercase letter **r**.
\b	Word boundary.

So this regular expression will match any string which can be described as "a word boundary, then a lowercase 'd', then zero or more word characters, then a lowercase 'r', then a word boundary." strings that can be described this way include the words **destroyer**, **dour**, and **doctor**, and the abbreviation **dr**.

To use this regular expression in Python search operations, we first compile it into a pattern object. For instance, the following Python statement creates a pattern object named **pattern** which we can use to perform searches using that regular expression.

```
pattern = re.compile(r"\bd\w*r\b")
```

Note

The letter **r** before our string in the statement above is important. It tells Python to interpret our string as a raw string, exactly as we've typed it. If we didn't prefix the string with an **r**, Python would interpret the escape sequences such as **\b** in other ways. Whenever you need Python to interpret your strings literally, specify it as a raw string by

prefixing it with **r**.

Now we can use the pattern object's methods, such as **search()** to search a string for the compiled regular expression, looking for a match. If it finds one, it will return a special result called a match object. Otherwise, it returns **None**, a built-in Python constant that is used like the boolean value "false".

Example

```
import re
str = "Good morning, doctor."
pat = re.compile(r"\bd\w*r\b")  # compile regex "\bd\w*r\b" to a
pattern object
if pat.search(str) != None:     # Search for the pattern. If
found,
    print("Found it.")
```

Output

```
Found it.
```

Example

To perform a case-insensitive search, you can specify the special constant **re.IGNORECASE** in the compile step:

```
import re
str = "Hello, DoctoR."
pat = re.compile(r"\bd\w*r\b", re.IGNORECASE)  # upper and
lowercase will match
if pat.search(str) != None:
    print("Found it.")
```

Output

```
Found it.
```

Putting it all together

So now we know how to open a file, read the lines into a list, and locate a substring in any given element of that list. Let's use this knowledge to build some example programs.

Print all lines containing substring

The program below reads a log file line by line. If the line contains the word "error," it is added to a list called **errors**. If not, it is ignored. The **lower()** string method converts all strings to lowercase for comparison purposes, making the search case-insensitive without altering the original strings.

Note that the **find()** method is called directly on the result of the **lower()** method; this is called *method chaining*. Also, note that in the **print()** statement, we construct an output string by joining several strings with the + operator.

Example

```
errors = []                       # The list where we will store
results.
linenum = 0
substr = "error".lower()          # Substring to search for.
with open ('logfile.txt', 'rt') as myfile:
    for line in myfile:
        linenum += 1
        if line.lower().find(substr) != -1:    # if case-
insensitive match,
            errors.append("Line " + str(linenum) + ": " +
line.rstrip('\n'))
for err in errors:
    print(err)
```

Output

```
Line 6: Mar 28 09:10:37 Error: cannot contact server. Connection
refused.
Line 10: Mar 28 10:28:15 Kernel error: The specified location is
not mounted.
Line 14: Mar 28 11:06:30 ERROR: usb 1-1: can't set config,
exiting.
```

Extract all lines containing substring, using regex

The program below is similar to the above program, but using the **re** regular expressions module. The errors and line numbers are stored as tuples, e.g., (linenum, line). The tuple is created by the additional enclosing parentheses in the **errors.append()** statement. The elements of the tuple are referenced similar to a list, with a zero-based index in brackets. As constructed here, **err[0]** is a linenum and **err[1]** is the associated line containing an error.

Example

```
import re
errors = []
linenum = 0
pattern = re.compile("error", re.IGNORECASE)  # Compile a case-
insensitive regex
with open ('logfile.txt', 'rt') as myfile:
```

```
    for line in myfile:
        linenum += 1
        if pattern.search(line) != None:      # If a match is
found
            errors.append((linenum, line.rstrip('\n')))
for err in errors:                            # Iterate over the
list of tuples
    print("Line " + str(err[0]) + ": " + err[1])
```

Output (same as above)

```
Line 6: Mar 28 09:10:37 Error: cannot contact server. Connection
refused.
Line 10: Mar 28 10:28:15 Kernel error: The specified location is
not mounted.
Line 14: Mar 28 11:06:30 ERROR: usb 1-1: can't set config,
exiting.
```

Extract all lines containing a phone number

The program below prints any line of a text file, **info.txt**, which contains a US or international phone number. It accomplishes this with the regular expression "**(\+\d{1,2})?[\s.-]?\d{3}[\s.-]?\d{4}**". This regex matches the following phone number notations:

- 123-456-7890
- (123) 456-7890
- 123 456 7890
- 123.456.7890
- +91 (123) 456-7890

```
import re
errors = []
linenum = 0
pattern = re.compile(r"(\+\d{1,2})?[\s.-]?\d{3}[\s.-]?\d{4}")
with open ('info.txt', 'rt') as myfile:
    for line in myfile:
        linenum += 1
        if pattern.search(line) != None:  # If pattern search
finds a match,
            errors.append((linenum, line.rstrip('\n')))
for err in errors:
    print("Line ", str(err[0]), ": " + err[1])
```

Output

```
Line  3 : My phone number is 731.215.8881.
Line  7 : You can reach Mr. Walters at (212) 558-3131.
Line  12 : His agent, Mrs. Kennedy, can be reached at +12 (123)
456-7890
Line  14 : She can also be contacted at (888) 312.8403, extension
12.
```

Search a dictionary for words

The program below searches the dictionary for any words that start with **h** and end in **pe**. For input, it uses a dictionary file included on many Unix systems, **/usr/share /dict/words**.

```
import re
filename = "/usr/share/dict/words"
pattern = re.compile(r"\bh\w*pe$", re.IGNORECASE)
with open(filename, "rt") as myfile:
    for line in myfile:
        if pattern.search(line) != None:
            print(line, end='')
```

Output

```
Hope
heliotrope
hope
hornpipe
horoscope
hype
```