

Debugger and Profiler

Regular Expression



#9
Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

Using the debugger is a good way to figure out what is causing bugs in your program. As an example, here is a small program that has a bug in it. The program comes up with a random addition problem for the user to solve.

```
import random

number1 = random.randint(1, 10)
number2 = random.randint(1, 10)
print('What is ' + str(number1) + ' + ' + str(number2) + '?')
answer = input()
if answer == (number1 + number2):
    print('Correct!')
else:
    print('Nope! The answer is ' + str(number1 + number2))
```

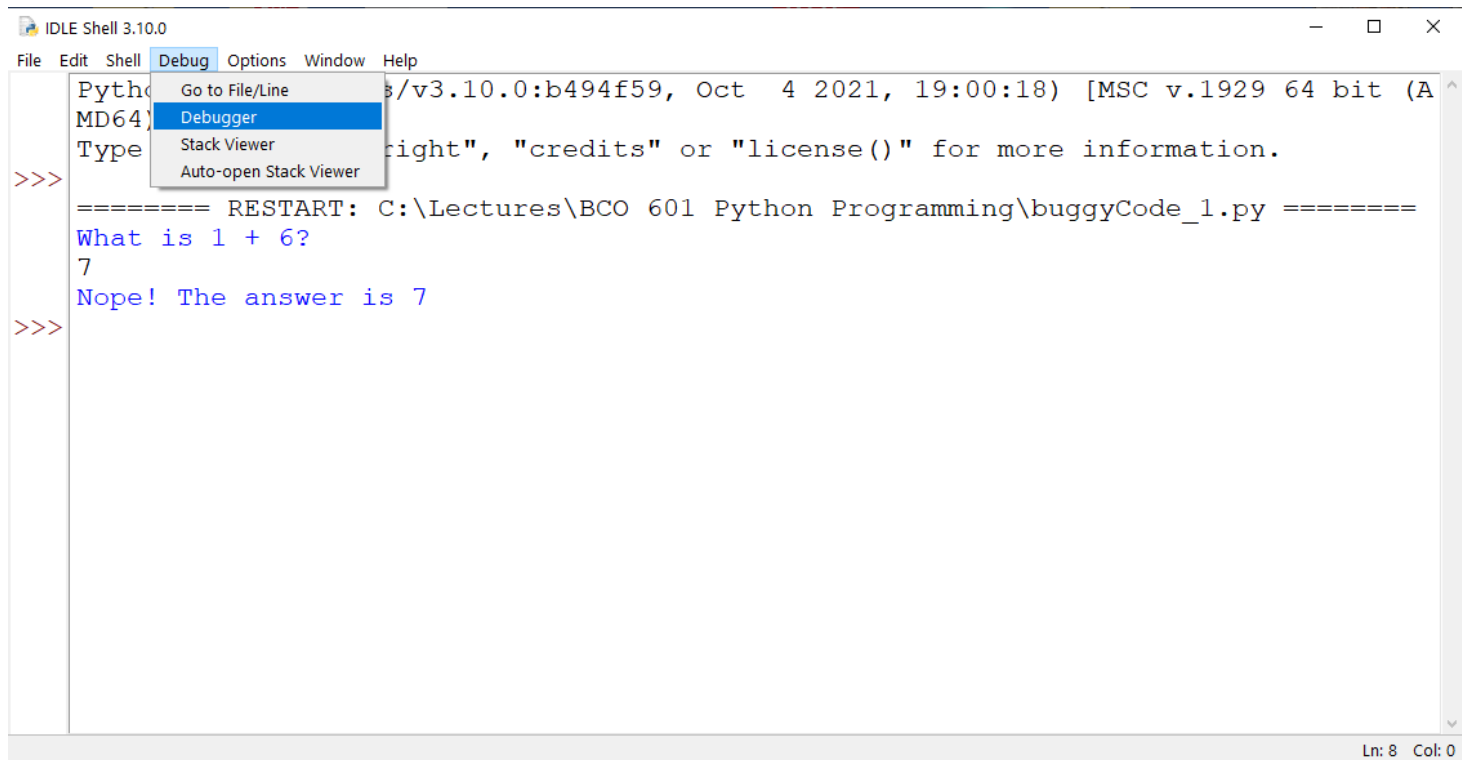
This is a simple arithmetic game that comes up with two random numbers and asks you to add them. Here's what it might look like when you run the program:

```
What is 5 + 1?
```

```
6
```

```
Nope! The answer is 6
```

That's not right! This program has a semantic bug in it. Even if the user types in the correct answer, the program says they are wrong. You could look at the code and think hard about where it went wrong. That works sometimes. But you might figure out the cause of the bug quicker if you run the program under the debugger.

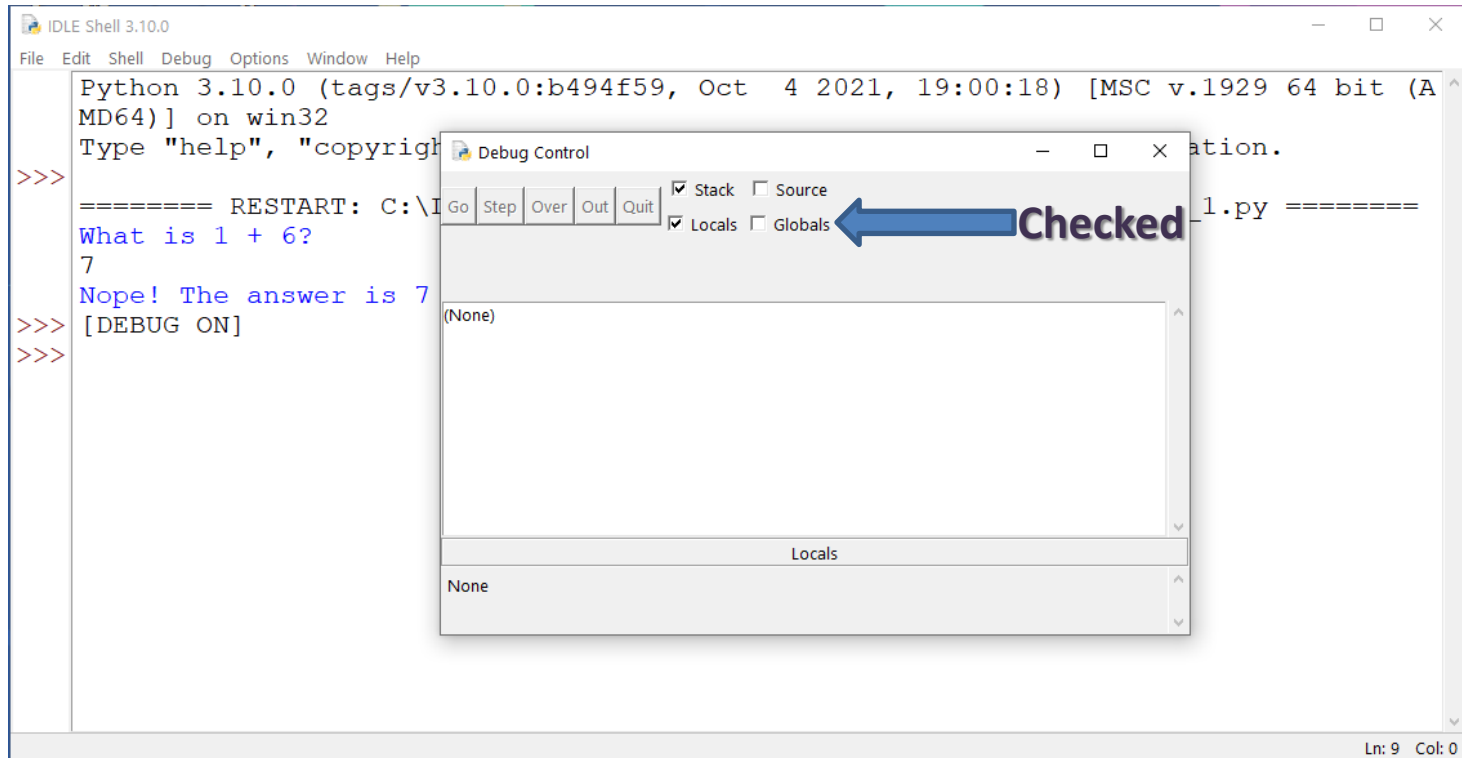


The screenshot shows the IDLE Shell 3.10.0 window. The 'Debug' menu is open, and the 'Debugger' option is highlighted. The shell contains the following text:

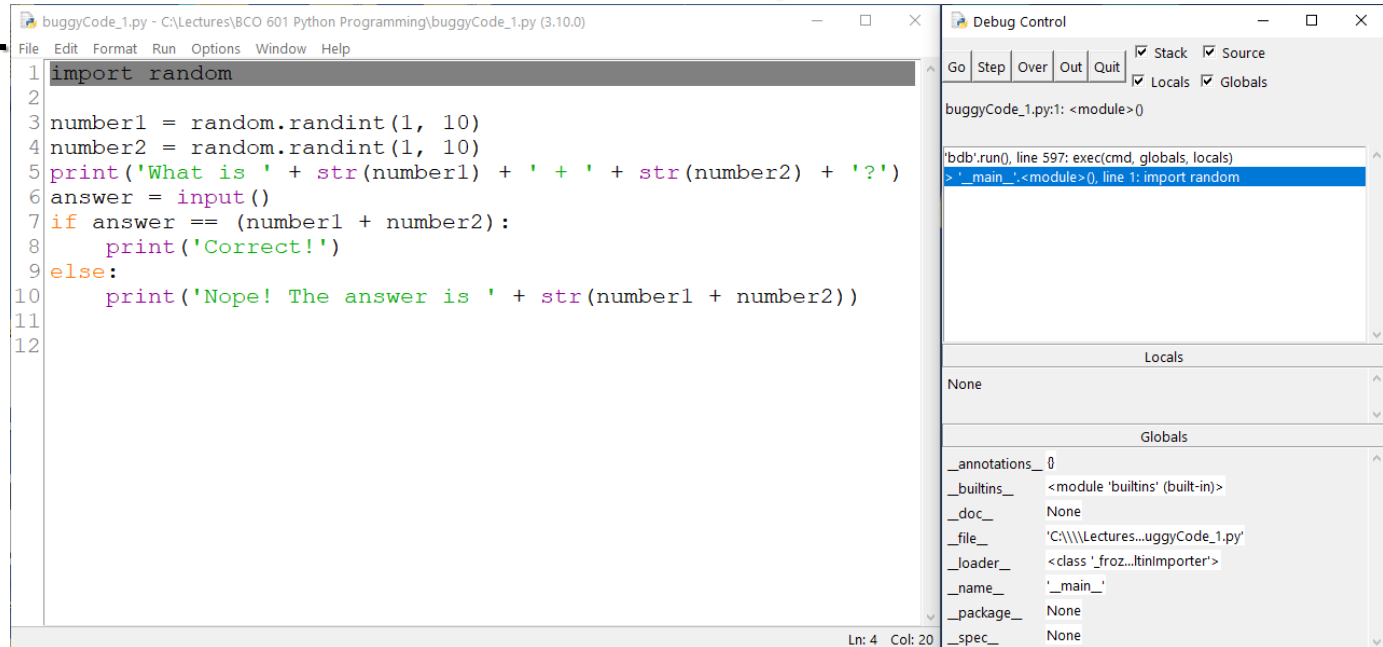
```
>>> Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)]
Type "help()", "copyright()", "credits()", "license()", "help()" or "help()" for more information.

>>> ===== RESTART: C:\Lectures\BCO 601 Python Programming\buggyCode_1.py =====
>>> What is 1 + 6?
>>> 7
>>> Nope! The answer is 7
```

The status bar at the bottom right indicates 'Ln: 8 Col: 0'.

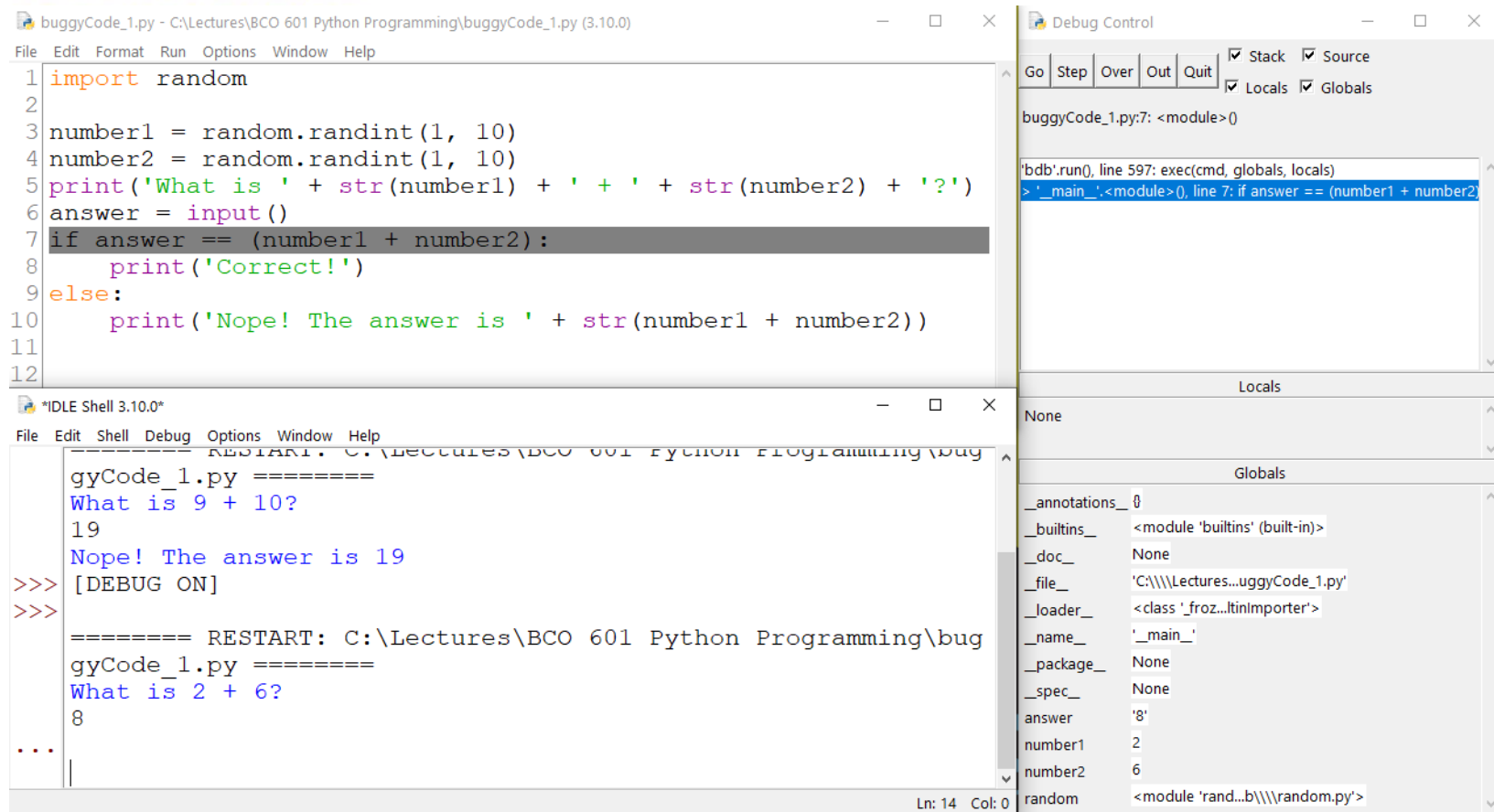


The debugger starts at the `import random` line. Nothing special happens here, so just click **Step** to execute it. You should see the random module at the bottom of the Debug Control window in the Globals area.



PYTHON PROGRAMMING

Using Debugger



buggyCode_1.py - C:\Lectures\BCO 601 Python Programming\buggyCode_1.py (3.10.0)

```
File Edit Format Run Options Window Help
1 import random
2
3 number1 = random.randint(1, 10)
4 number2 = random.randint(1, 10)
5 print('What is ' + str(number1) + ' + ' + str(number2) + '?')
6 answer = input()
7 if answer == (number1 + number2):
8     print('Correct!')
9 else:
10     print('Nope! The answer is ' + str(number1 + number2))
11
12
```

Debug Control

Go Step Over Out Quit ☒ Stack ☒ Source ☒ Locals ☒ Globals

buggyCode_1.py:7: <module>()>

'bdb'.run0, line 597: exec(cmd, globals, locals)

> '_main_'.<module>0, line 7: if answer == (number1 + number2)

Locals

None

Globals

__annotations__ []

__builtins__ <module 'builtins' (built-in)>

__doc__ None

__file__ 'C:\\\\Lectures...uggyCode_1.py'

__loader__ <class 'froz...ltnImporter'>

__name__ '_main_'

__package__ None

__spec__ None

answer '8'

number1 2

number2 6

random <module 'rand...b\\\\random.py'>

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
===== RESTART: C:\Lectures\BCO 601 Python Programming\bug
gyCode_1.py =====
What is 9 + 10?
19
Nope! The answer is 19
>>> [DEBUG ON]
>>>

===== RESTART: C:\Lectures\BCO 601 Python Programming\bug
gyCode_1.py =====
What is 2 + 6?
8
...
|
```

Ln: 14 Col: 0



```
# another buggy code
my_list = [5, 2, 1, True, "abcdefg", 3, False, 4]

del my_list[3]
del my_list[4]
del my_list[6] # this is the bug

print(my_list)
```

```
Traceback (most recent call last):
  File "D:/Lectures/BCO 601 Python Programming/buggy_another.py",
line 6, in <module>
    del my_list[6] # this is the bug
IndexError: list assignment index out of range
```



one solution to check where the bug is to use print

```
my_list = [5, 2, 1, True, "abcdfg", 3, False, 4]
```

```
del my_list[3]
```

```
→ print(my_list)
```

```
del my_list[4]
```

```
→ print(my_list)
```

```
del my_list[6]
```

```
→ print(my_list)
```



```
# using pdb
import pdb
my_list = [5, 2, 1, True, "abcdefg", 3, False, 4]

pdb.set_trace()
del my_list[3]
del my_list[4]
del my_list[6]

> buggy_another.py(6)<module>()
-> del my_list[3]
(Pdb)
```





```
buggyCode_2.py - C:\Lectures\BCO 601 Python Programming\buggyCode_2.py (3.13.0)
File Edit Format Run Options Window Help

1 import pdb
2
3 my_list = [5, 2, 1, True, "abcdefg", 3, False, 4]
4
5 pdb.set_trace()
6 del my_list[3]
7 del my_list[4]
8 del my_list[6]
9

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help

Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1914 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
[DEBUG ON]
>>>
>>>
===== RESTART: C:\Lectures\BCO 601 Python Programming\buggyCode_1.py ==
=====
What is 3 + 1?
4
Nope! The answer is 4
>>> [DEBUG ON]
>>> [DEBUG OFF]
>>>
===== RESTART: C:\Lectures\BCO 601 Python Programming\buggyCode_2.py =
=====
> c:\lectures\bco 601 python programming\buggycode_2.py(5) <module>()
-> pdb.set_trace()
(Pdb)
```

Ln: 15 Col: 6



```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcdefg", 3, False, 4]
4
5 pdb.set_trace()
6 del my_list[3]
7 del my_list[4]
8 del my_list[6]

> buggy_another.py(6)<module>()
-> del my_list[3]
(Pdb)
```




```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
5 pdb.set_trace()
➔ 6 del my_list[3]
7 del my_list[4]
8 del my_list[6]
```

```
> buggy_another.py(6)<module>()
-> del my_list[3]
(Pdb) my_list
[5, 2, 1, True, 'abcfg', 3, False, 4]
(Pdb)
```



```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
5 pdb.set_trace()
➔ 6 del my_list[3]
7 del my_list[4]
8 del my_list[6]
```

```
> buggy_another.py(6)<module>()
-> del my_list[3]
(Pdb) my_list
[5, 2, 1, True, 'abcfg', 3, False, 4]
(Pdb) next      # or you can type n instead.
```



```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
5 pdb.set_trace()
6 del my_list[3]
7 del my_list[4]
8 del my_list[6]
```

```
> buggy_another.py(6)<module>()
-> del my_list[3]
(Pdb) my_list
[5, 2, 1, True, 'abcfg', 3, False, 4]
(Pdb) n
> buggy_another.py(7)<module>()
-> del my_list[4]
```



```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
5 pdb.set_trace()
6 del my_list[3]
7 del my_list[4]
8 del my_list[6]
```

```
> buggy_another.py(7)<module>()
-> del my_list[4]
(Pdb) my_list
[5, 2, 1, 'abcfg', 3, False, 4]
(Pdb) n
> buggy_another.py(8)<module>()
-> del my_list[6]
```



```
1 # using pdb
2 import pdb
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
5 pdb.set_trace()
6 del my_list[3]
7 del my_list[4]
➡ 8 del my_list[6]
```

```
> buggy_another.py(7)<module>()
```

```
-> del my_list[6]
```

```
(Pdb) my_list
```

```
[5, 2, 1, 'abcfg', False, 4]
```

```
(Pdb) c          # continue
```

```
Traceback (most recent call last):
```

```
IndexError: list assignment index out of range
```



```
1 # using pdb
2
3 my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
4
➔ 5 import pdb; pdb.set_trace() # This is the only time you can use ;
6
7 del my_list[3]
8 del my_list[4]
9 del my_list[6]
```

Make debugging options in one line command



Starting in Python 3.7, there's another way to enter the debugger. **PEP 553** describes the built-in function `breakpoint()`, which makes entering the debugger easy and consistent:

```
breakpoint()
```

By default, `breakpoint()` will `import pdb` and call `pdb.set_trace()`, as shown above. However, using `breakpoint()` is more flexible and allows you to control debugging behavior via its API and use of the environment variable `PYTHONBREAKPOINT`.

For example, setting `PYTHONBREAKPOINT=0` in your environment will completely disable `breakpoint()`, thus disabling debugging.

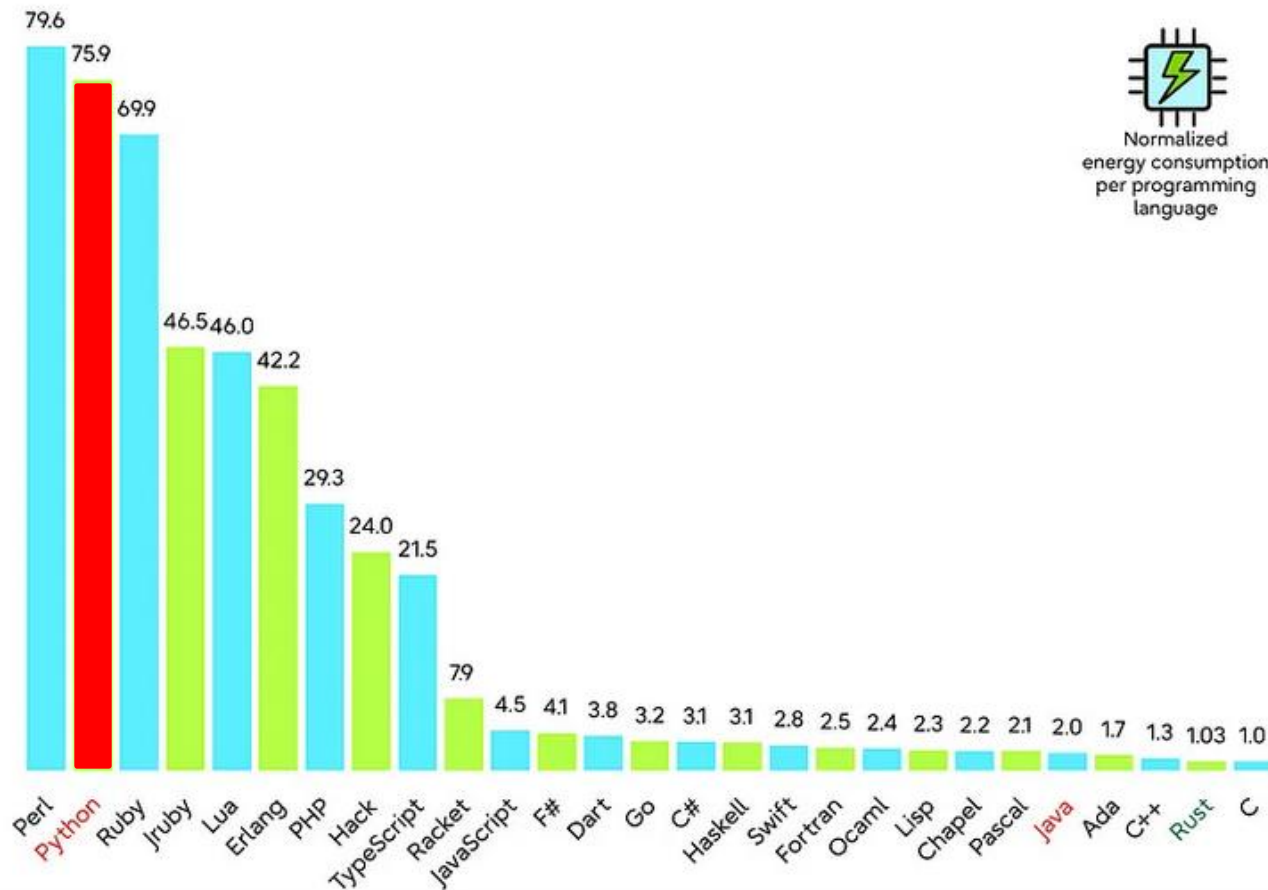


```
my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
breakpoint()
del my_list[3]
del my_list[4]
del my_list[6]
```

```
-> del my_list[3]
(Pdb) p my_list
[5, 2, 1, True, 'abcfg', 3, False, 4]
(Pdb) n
-> del my_list[4]
(Pdb) l
1      my_list = [5, 2, 1, True, "abcfg", 3, False, 4]
2      breakpoint()
3      del my_list[3]
4  -> del my_list[4]
5      del my_list[6]
```


Command	Description
p	Print the value of an expression.
pp	Pretty-print the value of an expression.
n	Continue execution until the next line in the current function is reached or it returns.
s	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).
c	Continue execution and only stop when a breakpoint is encountered.
unt	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
l	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
ll	List the whole source code for the current function or frame.
b	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
w	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

Using Profiler



```
import profile
import re
```

```
profile.run('re.compile("foo|bar")')
```

221 function calls (214 primitive calls) in 0.016 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
48	0.000	0.000	0.000	0.000	:0(append)
1	0.000	0.000	0.000	0.000	:0(compile)
1	0.000	0.000	0.000	0.000	:0(exec)
5	0.000	0.000	0.000	0.000	:0(find)
29	0.000	0.000	0.000	0.000	:0(isinstance)
1	0.000	0.000	0.000	0.000	:0(items)
30/27	0.000	0.000	0.000	0.000	:0(len)

The first line indicates that 221 calls were monitored. Of those calls, 214 were primitive, meaning that the call was not induced via recursion.

ncalls : for the number of calls.

Tottime : for the total time spent in the given function (and excluding time made in calls to sub-functions)

Percall : is the quotient of `tottime` divided by `ncalls`

cumtime : is the cumulative time spent in this and all subfunctions (from invocation till exit). this figure is accurate even for recursive functions.

Percall : is the quotient of **cumtime** divided by primitive calls

filename: lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of `primitive calls` and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

PYTHON PROGRAMMING

Using profile

```
import profile
```

```
def fib(n):  
    return n if n < 2 else fib(n - 2) + fib(n - 1)
```

```
print(fib(35))  
profile.run('fib(35)', sort='tottime')
```

29860707 function calls (5 primitive calls) in 44.094 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
29860703/1	44.094	0.000	44.094	44.094	profileCode_2.py:5(fib)
0	0.000		0.000		profile:0(profiler)
1	0.000	0.000	44.094	44.094	profile:0(fib(35))
1	0.000	0.000	44.094	44.094	:0(exec)
1	0.000	0.000	44.094	44.094	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	:0(setprofile)

PYTHON PROGRAMMING

Using profile

```
import profile
```

```
def fib(n):  
    return n if n < 2 else fib(n - 2) + fib(n - 1)
```

```
profile.run('fib(35)')
```

9227465

29860707 function calls (5 primitive calls) in 42.734 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	42.719	42.719	:0(exec)
1	0.016	0.016	0.016	0.016	:0(setprofile)
1	0.000	0.000	42.719	42.719	<string>:1(<module>)
1	0.000	0.000	42.734	42.734	profile:0(fib(35))
0	0.000		0.000		profile:0(profiler)
29860703/1	42.719	0.000	42.719	42.719	profileCode_2.py:3(fib)



File and Text Operations

A ***file*** is a stream of text or bytes that a program can read and/or write; a ***filesystem*** is a hierarchical repository of files on a computer system.

Files and streams come in many flavors. Their contents can be arbitrary ***bytes***, or ***text***. They may be suitable for ***reading***, ***writing***, or both, and they may be ***buffered***, so that data is temporarily held in memory on the way to or from the file.

Files may also allow random access, moving forward and back within the file, or jumping to read or write at a particular location in the file.



To create a Python file object, call `open` with the following syntax:

```
open(file, mode='r', buffering=-1, encoding=None, errors='strict',  
      newline=None, closefd=True, opener=os.open)
```

file can be a string or an instance of `pathlib`.

Opening a File Pythonically

`open` is a context manager: **use with** `open(...)` **as** `f`.,

not `f = open(...)`,

to ensure the file `f` gets closed as soon as the with statement's body is done.



mode is an optional string indicating how the file is to be opened (or created).

'a'	The file is opened in write-only mode. The file is kept intact if it already exists, and the data you write is appended to the existing contents. The file is created if it does not exist. Calling <i>f.seek</i> on the file changes the result of the method <i>f.tell</i> , but does not change the write position in the file opened in this mode: that write position always remains at the end of the file.
'a+'	The file is opened for both reading and writing, so all methods of <i>f</i> can be called. The file is kept intact if it already exists, and the data you write is appended to the existing contents. The file is created if it does not exist. Calling <i>f.seek</i> on the file, depending on the underlying operating system, may have no effect when the next I/O operation on <i>f</i> writes data, but does work normally when the next I/O operation on <i>f</i> reads data.
'r'	The file must already exist, and it is opened in read-only mode (this is the default).
'r+'	The file must exist and is opened for both reading and writing, so all methods of <i>f</i> can be called.
'w'	The file is opened in write-only mode. The file is truncated to zero length and overwritten if it already exists, or created if it does not exist.
'w+'	The file is opened for both reading and writing, so all methods of <i>f</i> can be called. The file is truncated to zero length and overwritten if it already exists, or created if it does not exist.



File and Text Operations

The *mode* string may include any of the values in the previous slide, followed by a *b* or *t*.

b indicates that the file should be opened (or created) in **binary** mode, while *t* indicates **text** mode.

When neither *b* nor *t* is included, the default is text (i.e., '*r*' is like '*rt*', '*w+*' is like '*w+t*', and so on), but per The Zen of Python, “**explicit is better than implicit.**”



buffering is an integer value that denotes the buffering policy you're requesting for the file. When *buffering* is **0**, the file (which must be binary mode) is **unbuffered**; the effect is as if the file's buffer is flushed every time you write anything to the file.

When *buffering* is **1**, the file (which must be open in text mode) is line **buffered**, which means the file's buffer is flushed every time you write `\n` to the file. When buffering is greater than 1, the file uses a buffer of about *buffering* bytes, often rounded up to some value convenient for the driver software. When *buffering* is **<0**, a default is used, depending on the type of file stream. Normally, this default is line buffering for files that correspond to interactive streams, and a buffer of `io.DEFAULT_BUFFER_SIZE` bytes for other files.

Iteration on File Objects

A file object *f*, open for reading, is also an iterator whose items are the file's lines.

Thus, the loop:

```
for line in f:
```

iterates on each line of the file.

Due to buffering issues, interrupting such a loop prematurely (e.g., with *break*), or calling *next(f)* instead of *f.readline()*, leaves the file's position set to an arbitrary value.

Regular Expressions

HOW TO REGEX

STEP 1: OPEN YOUR FAVORITE EDITOR



STEP 2: LET YOUR CAT PLAY ON YOUR KEYBOARD



Regular Expressions

In computing, a regular expression, also referred to as "**regex**" or "**regexp**", provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor.

- Very powerful and quite cryptic
- Fun once you understand!! them
- Regular expressions are a language unto themselves
- A language of "marker characters" - programming with characters
- It is kind of an "**old school**" language - compact

Regular Expressions

.NET : The `System.Text.RegularExpressions` package provides various search-and replace functions.

Java : The `java.util.regex` package has built-in search-and-replace functions.

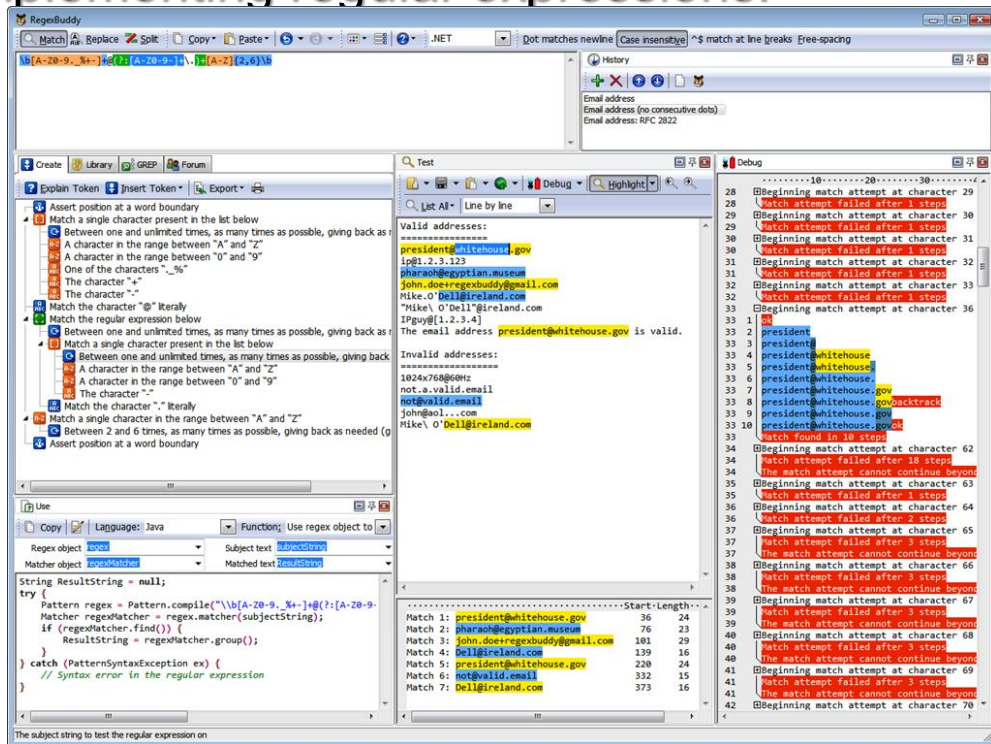
Perl : Perl has built-in support for regular expression substitution via the `s/regex/ replace/` operator.

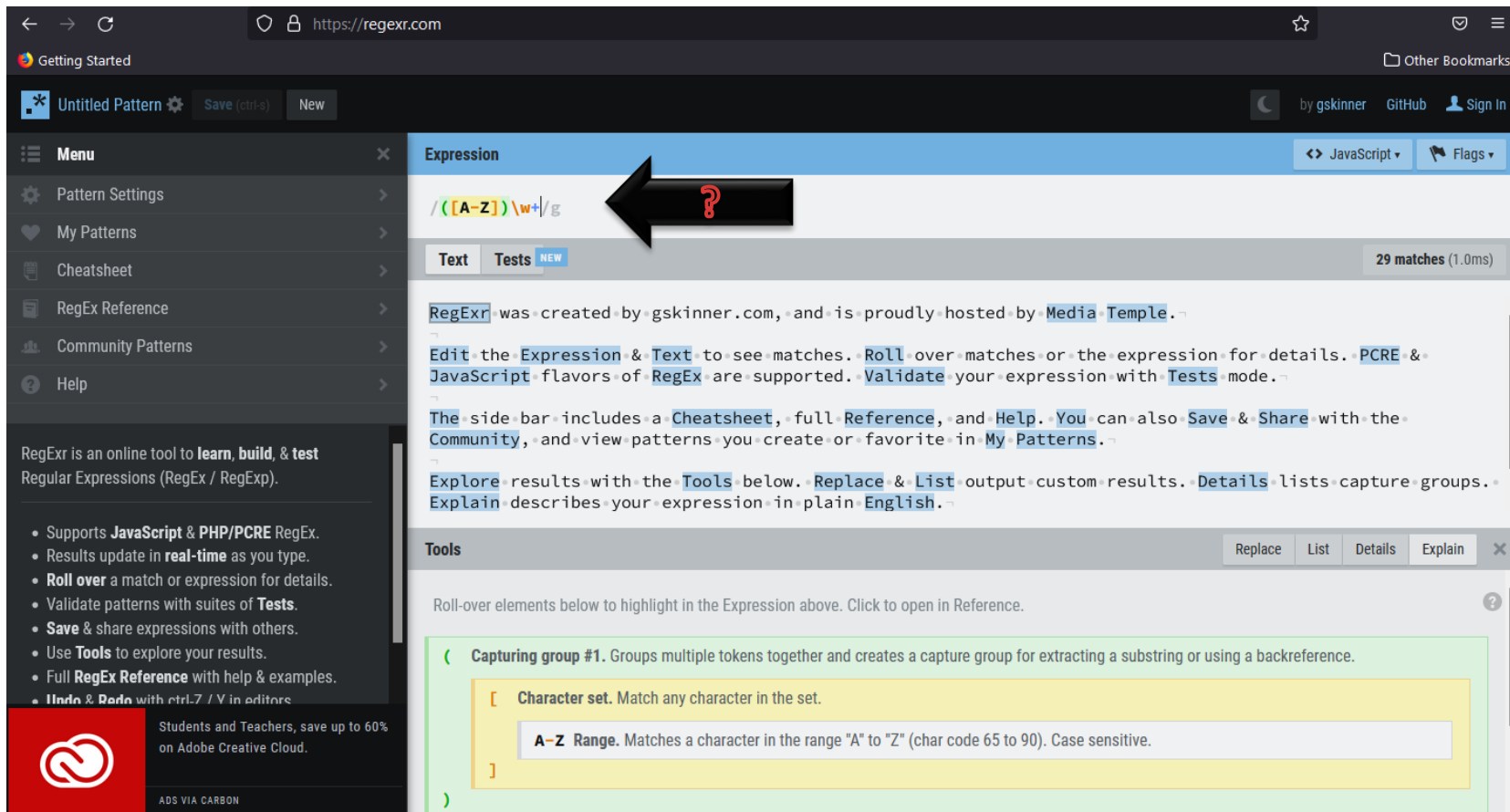
Python : Python's `re` module provides a sub function to search and replace.

Ruby : Ruby's regular expression support is part of the Ruby language itself, including the search-and-replace function.



RegexBuddy is the most full-featured tool available for creating, testing, and implementing regular expressions.





Getting Started

Untitled Pattern Save (ctrl-s) New

Menu

- Pattern Settings
- My Patterns
- Cheatsheet
- RegEx Reference
- Community Patterns
- Help

RegExr is an online tool to **learn, build, & test** Regular Expressions (RegEx / RegExp).

- Supports **JavaScript & PHP/PCRE** RegEx.
- Results update in **real-time** as you type.
- Roll over** a match or expression for details.
- Validate patterns with suites of **Tests**.
- Save & share** expressions with others.
- Use **Tools** to explore your results.
- Full **RegEx Reference** with help & examples.
- Undo & Redo** with ctrl-Z / Y in editors.

Students and Teachers, save up to 60% on Adobe Creative Cloud.

ADS VIA CARBON

Expression

`/([A-Z]\w+)/g`

JavaScript Flags

Text Tests **NEW** 29 matches (1.0ms)

RegExr was created by gskinner.com, and is proudly hosted by Media Temple.

Edit the Expression & Text to see matches. Roll over matches or the expression for details. PCRE & JavaScript flavors of RegEx are supported. Validate your expression with Tests mode.

The side bar includes a Cheatsheet, full Reference, and Help. You can also Save & Share with the Community, and view patterns you create or favorite in My Patterns.

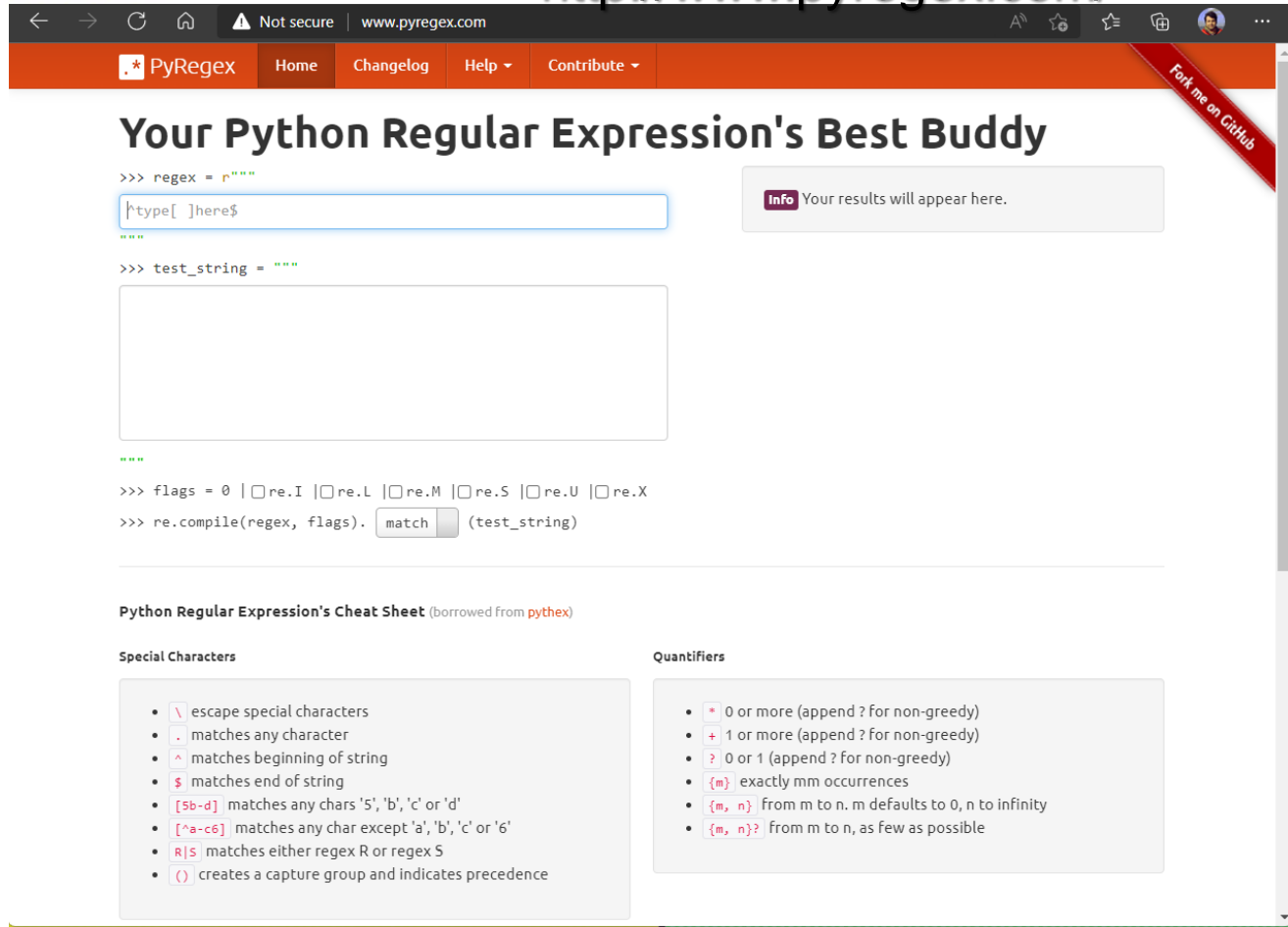
Explore results with the Tools below. Replace & List output custom results. Details lists capture groups. Explain describes your expression in plain English.

Tools

Replace List Details Explain

Roll-over elements below to highlight in the Expression above. Click to open in Reference.

- Capturing group #1.** Groups multiple tokens together and creates a capture group for extracting a substring or using a backreference.
- Character set.** Match any character in the set.
- A-Z Range.** Matches a character in the range "A" to "Z" (char code 65 to 90). Case sensitive.



The screenshot shows the PyRegex website. At the top is a navigation bar with links: Home, Changelog, Help, and Contribute. Below this is a main heading "Your Python Regular Expression's Best Buddy". There is a text input field containing the regex pattern `{type[]here$`. Below the input field is a large empty text area for the test string. To the right of the input field is an "Info" box stating "Your results will appear here." Below the text area are checkboxes for flags: `re.I`, `re.L`, `re.M`, `re.S`, `re.U`, and `re.X`. Below the flags is a "match" button and a text input for the test string. At the bottom of the page is a "Python Regular Expression's Cheat Sheet" section, which is divided into two columns: "Special Characters" and "Quantifiers".

PyRegex

Home Changelog Help Contribute

Your Python Regular Expression's Best Buddy

`>>> regex = r"{type[]here$"`

`>>> test_string = ""`

`>>> flags = 0 | ☐ re.I | ☐ re.L | ☐ re.M | ☐ re.S | ☐ re.U | ☐ re.X`

`>>> re.compile(regex, flags).match(test_string)`

match

Info Your results will appear here.

Python Regular Expression's Cheat Sheet (borrowed from pythex)

Special Characters

- `\` escape special characters
- `.` matches any character
- `^` matches beginning of string
- `$` matches end of string
- `[Sb-d]` matches any chars 'S', 'b', 'c' or 'd'
- `[^a-c6]` matches any char except 'a', 'b', 'c' or '6'
- `R|S` matches either regex R or regex S
- `()` creates a capture group and indicates precedence

Quantifiers

- `*` 0 or more (append ? for non-greedy)
- `+` 1 or more (append ? for non-greedy)
- `?` 0 or 1 (append ? for non-greedy)
- `{m}` exactly mm occurrences
- `{m, n}` from m to n. m defaults to 0, n to infinity
- `{m, n}?` from m to n, as few as possible



^	Matches the beginning of a line
\$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*	Repeats a character zero or more times
*?	Repeats a character zero or more times (non-greedy)
+	Repeats a character one or more times
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(Indicates where string extraction is to start
)	Indicates where string extraction is to end

Regular Expressions

Before you can use regular expressions in your program, you must import the library using `"import re"`

You can use `re.search()` to see if a string matches a regular expression similar to using the `find()` method for strings

You can use `re.findall()` extract portions of a string that match your regular expression similar to a combination of `find()` and slicing: `var[5:10]`

For quick one-off tests, you can use the global function

```
if re.search("regex pattern", subject):  
    # Successful match  
else:  
    # Match attempt failed
```

To use the same regex repeatedly, use a compiled object

```
reobj = re.compile("regex pattern")  
if reobj.search(subject):  
    # Successful match  
else:  
    # Match attempt failed
```



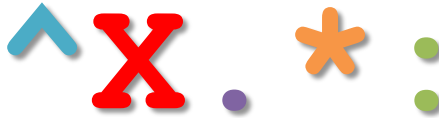
```
filename = 'alice.txt'
try:
    with open(filename, encoding='utf-8') as f:
        contents = f.readlines()
except FileNotFoundError:
    print(f'Sorry, the file {filename} does not exist.')
else:
    for line in contents:
        line = line.rstrip()
        if line.find('"poison,"') >= 0:
            print(line)
```



```
filename = 'alice.txt'
try:
    with open(filename, encoding='utf-8') as f:
        contents = f.readlines()
except FileNotFoundError:
    print(f'Sorry, the file {filename} does not exist.')
else:
    for line in contents:
        line = line.rstrip()
        if re.search('"poison"', line):
            print(line)
```

Wild-Card Characters

- The **dot** character matches any character
- If you add the **asterisk** character, the character is “any number of times”



Wild-Card Characters

Expression JavaScript Flags

`/^X.*:/g`

Text Tests 1 match (0.0ms)

X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: InnocentX-DSPAM-Confidence: 0.8475X-Content-Type-Message-Body: text/plain

Tools Replace List Details Explain ×

^ Beginning. Matches the beginning of the string, or the beginning of a line if the multiline flag (m) is enabled.

X Character. Matches a "X" character (char code 88). Case sensitive.

. Dot. Matches any character except line breaks.

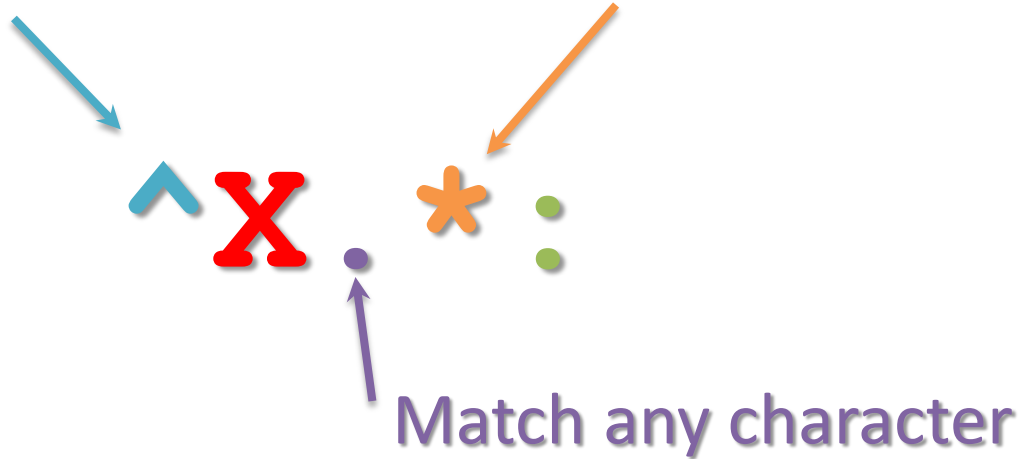
***** Quantifier. Match 0 or more of the preceding token.

: Character. Matches a ":" character (char code 58).

Wild-Card Characters

Match the start of the line

Many times



X-Sieve: CMU Sieve 2.3

X-DSPAM-Result: InnocentX-DSPAM-Confidence: 0.8475X-

Content-Type-Message-Body: text/plain



Wild-Card Characters

The `re.search()` returns a **True/False** depending on whether the string matches the regular expression

If we actually want the matching strings to be extracted, we use `re.findall()`

[0-9]+



One or more digits

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
>>> ['2', '19', '42']
```

When we use `re.findall()` it returns a list of zero or more substrings that match the regular expression

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
>>> y = re.findall('[AEIOU]+', x)
>>> print(y)
>>> []
```



Warning: Greedy Matching

The **repeat** characters (***** and **+**) push outward in both directions (greedy) to match the largest possible string

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+:', x)
>>> print(y)
>>> ['From: Using the :']
```

Expecting 'From:' ?

One or more characters

First character in the match is an F

Last character in the match is a :



Warning: Greedy Matching

Not all regular expression repeat codes are **greedy**! If you add a ? character - the + and *

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print(y)
>>> ['From:']
```

One or more characters

First character in the match is an F

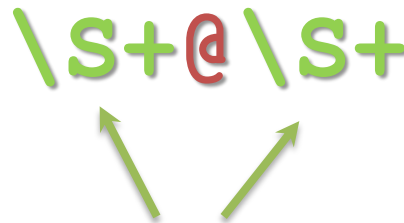
Last character in the match is a :

Fine Tuning String Extraction

You can refine the match for `re.findall()` and separately determine which portion of the match that is to be extracted using parenthesis

From **stephen.marquard@uct.ac.uk** **Sat Jan 5 09:14:16 2008**

```
>>> y = re.findall('\S+@\S+', x)
>>> print(y)
['stephen.marquard@uct.ac.uk']
>>> y = re.findall('^From:.*? (\S+@\S+)', x)
>>> print(y)
['stephen.marquard@uct.ac.uk']
```



`\S+@\S+`

At least one non-whitespace character

Parenthesis are not part of the match - but they tell where to start and stop what string to extract

From **stephen.marquard@uct.ac.uk** Sat Jan 5 09:14:16 2008

```
>>> y = re.findall('\S+@\S+', x)
>>> print(y)
['stephen.marquard@uct.ac.uk']
>>> y = re.findall('^From:.*? (\S+@\S+)', x)
>>> print(y)
['stephen.marquard@uct.ac.uk']
```

^From (\S+@ \S+)



```
import re

txt = """
xenarthral xerically xenomorphically xebec xenomania
xenogenic xenogeny xenophobically xenon xenomenia
xylotomy xenogenies xenografts xeroxing xenons xanthous
xenoglossy xanthopterins xenoglossy xeroxed xenophoby
xenoglossies xanthoxyls xenoglossias xenomorphically
xeroxes xanthopterin xebecs xenodochiums xenodochium
xylopyrography xanthopterines xerochasy xenium xenic
"""

pat1 = re.compile(r'x.*y')          # greedy quantifier

for i, match in enumerate(re.findall(pat1, txt), 1):
    print(str(i) + ' ' + match)
```



xenarthral xerically xenomorphically xebec xenomania
xenogenic xenogeny xenophobically xenon xenomenia
xylotomy xenogenies xenografts xeroxing xenons xanthous
xenoglossy xanthopterins xenoglossy xeroxed xenophoby
xenoglossies xanthoxyls xenoglossias xenomorphically
xerxes xanthopterin xebecs xenodochiums xenodochium
xylopyrography xanthopterines xerochasy xenium xenic

```
pat1 = re.compile(r'x.*y')      # greedy quantifier
```

- 1.xenarthral xerically xenomorphically
- 2.xenogenic xenogeny xenophobically
- 3.xylotomy
- 4.xenoglossy xanthopterins xenoglossy xeroxed xenophoby
- 5.xenoglossies xanthoxyls xenoglossias xenomorphically
- 6.xylopyrography xanthopterines xerochasy



xenarthral xerically xenomorphically xebec xenomania
xenogenic xenogeny xenophobically xenon xenomenia
xylotomy xenogenies xenografts xeroxing xenons xanthous
xenoglossy xanthopterins xenoglossy xeroxed xenophobia
xenoglossies xanthoxyls xenoglossias xenomorphically
xerxes xanthopterin xebecs xenodochiums xenodochium
xylopyrography xanthopterines xerochasy xenium xenic

```
pat2 = re.compile(r'x.*?y')      # non-greedy quantifier
```

```
1.xenarthral xerically
2.xenomorphically
3.xenogenic xenogeny
4.xenophobically
5.xy
6.xenoglossy
7.xanthopterins xenoglossy
8.xeroxed xenophobia
9.xenoglossies xanthoxy
10.xenoglossias xenomorphically
11.xy
12.xanthopterines xerochasy
```



xenarthral xerically xenomorphically xebec xenomania
xenogenic xenogeny xenophobically xenon xenomenia
xylotomy xenogenies xenografts xeroxing xenons xanthous
xenoglossy xanthopterins xenoglossy xeroxed xenophobia
xenoglossies xanthoxyls xenoglossias xenomorphically
xeroxes xanthopterin xebecs xenodochiums xenodochium
xylopyrography xanthopterines xerochasy xenium xenic

```
pat3 = re.compile(r'x[a-z]*y')
```

- 1.xerically
- 2.xenomorphically
- 3.xenogeny
- 4.xenophobically
- 5.xylotomy
- 6.xenoglossy
- 7.xenoglossy
- 8.xenophobia
- 9.xanthoxy
- 10.xenomorphically
- 11.xylopyrography
- 12.xerochasy

```
txt_short = "breathiness xenogeny randed xyxyblah xyotomy"
```

```
for i, match in enumerate(re.findall(pat3, txt_short), 1):  
    print(str(i) + "." + match)
```

```
1.xenogeny  
2.xyxy ← The prefix xyxy is not a full word  
3.xyotomy
```

The easiest is to use the explicit “**word boundary**” special zero-width match pattern, spelled as `\b` in Python and many other regular expression engines:

```
pat4 = re.compile(r"\b[x][a-z]*[y]\b")
```

```
for i, match in enumerate(re.findall(pat4, txt_short), 1):  
    print(str(i) + "." + match)
```

```
1.xenogeny  
2.xyotomy
```

Fine Tuning String Extraction

21 31



From **stephen.marquard@uct.ac.uk** Sat Jan 5 09:14:16 2008

```
>>> data = 'From stephen.marquard@uct.ac.uk Sat Jan 5  
09:14:16 2008'
```

```
>>> atpos = data.find('@')
```

```
>>> print(atpos)
```

```
>>> 21
```

```
>>> spos = data.find(' ', atpos)
```

```
>>> print(spos)
```

```
>>> 31
```

```
>>> host = data[atpos+1 : spos]
```

```
>>> print(host)
```

```
>>> uct.ac.uk
```

Extracting a host name -
using find and string slicing.

Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

From **stephen.marquard@uct.ac.uk** **Sat Jan 5 09:14:16 2008**

```
>>> line = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> words = line.split()
>>> print(words)
['From', 'stephen.marquard@uct.ac.uk', 'Sat', 'Jan', '5',
'09:14:16', '2008']
>>> email = words[1]
>>> pieces = email.split('@')
>>> print(pieces[1])
```

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('@([ ^ ]*)', lin)
>>> print(y)
['uct.ac.uk']
```

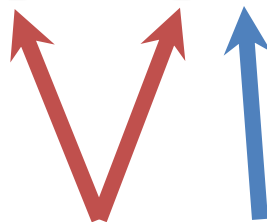
' @ ([^] *) '



Look through the string until you find an at-sign


```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('@([ ^ ]*)', lin)
>>> print(y)
['uct.ac.uk']
```

' @ ([^] *) '




Match non-blank character

Match many of them

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('@([ ^ ]*)', lin)
>>> print(y)
['uct.ac.uk']
```

' @ ([^] *) '



Extract the non-blank characters

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('^From.*@([ ^ ]*)', lin)
>>> print(y)
['uct.ac.uk']
```

 **^From.*@([^]*)'**

Starting at the beginning of the line, look for the string 'From '

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('^From.*@([ ^]*)', lin)
>>> print(y)
['uct.ac.uk']
```

`\^From.*@([^]*)'`

An orange arrow points from the text 'Skip a bunch of characters, looking for an at-sign' to the asterisk in the regex pattern. A blue arrow points from the same text to the at-sign in the regex pattern.

Skip a bunch of characters, looking for an at-sign

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('^From.*@([ ^ ]*)', lin)
>>> print(y)
['uct.ac.uk']
```

`\^From.*@([^]*)'`

Start 'extracting'

Stop 'extracting'

```
>>> import re
>>> lin = 'From stephen.marquard@uct.ac.uk Sat Jan 5
09:14:16 2008'
>>> y = re.findall('^From.*@([ ^]*)', lin)
>>> print(y)
['uct.ac.uk']
```

`^From.*@([^]*)`

Match non-blank character

Match many of them

Regex Metacharacters

Character	Description	Example
<code>[]</code>	A set of characters	<code>"[a-m]"</code>
<code>\</code>	Signals a special sequence (can also be used to escape special characters)	<code>"\d"</code>
<code>.</code>	Any character (except newline character)	<code>"he..o"</code>
<code>^</code>	Starts with	<code>"^hello"</code>
<code>\$</code>	Ends with	<code>"world\$"</code>
<code>*</code>	Zero or more occurrences	<code>"aix*"</code>
<code>+</code>	One or more occurrences	<code>"aix+"</code>
<code>{}</code>	Exactly the specified number of occurrences	<code>"al{2}"</code>
<code> </code>	Either or	<code>"falls stays"</code>
<code>()</code>	Capture and group	

Regex Special Sequences

Character	Description	Example
<code>\A</code>	Returns a match if the specified characters are at the beginning of the string	<code>"\AThe"</code>
<code>\b</code>	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	<code>r"\bain"</code> <code>r"ain\b"</code>
<code>\B</code>	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	<code>r"\Bain"</code> <code>r"ain\B"</code>
<code>\d</code>	Returns a match where the string contains digits (numbers from 0-9)	<code>"\d"</code>
<code>\D</code>	Returns a match where the string DOES NOT contain digits	<code>"\D"</code>
<code>\s</code>	Returns a match where the string contains a white space character	<code>"\s"</code>
<code>\S</code>	Returns a match where the string DOES NOT contain a white space character	<code>"\S"</code>
<code>\w</code>	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore <code>_</code> character)	<code>"\w"</code>
<code>\W</code>	Returns a match where the string DOES NOT contain any word characters	<code>"\W"</code>
<code>\Z</code>	Returns a match if the specified characters are at the end of the string	<code>"Spain\Z"</code>

Regex Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning

Set	Description
[am]	Returns a match where one of the specified characters (a , r , or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^am]	Returns a match for any character EXCEPT a , r , and n
[0123]	Returns a match where any of the specified digits (0 , 1 , 2 , or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z , lower case OR upper case
[+]	In sets, + , * , . , , () , \$, {} has no special meaning, so [+] means: return a match for any + character in the string

```
import re
```

```
first5letters = input('enter only letters a to e ')  
if not re.match('^[abcde]*$', first5letters):  
    print("Error! Only letters (a , b, c, d, e) are allowed!")  
else:  
    print(f'You entered : {first5letters}')
```

```
enter only letters a to e merhaba
```

```
Error! Only letters (a , b, c, d, e) are allowed!
```

```
enter only letters a to e baba
```

```
You entered : baba
```

```
enter only letters a to e BABA
```

```
Error! Only letters (a , b, c, d, e) are allowed!
```

```
import re
```

```
txt = "The rain in Ankara"
```

```
# The sub() function replaces the matches with the text
```

```
x = re.sub("\s", ".", txt)
```

```
print(x)
```

```
The.rain.in.Ankara
```

```
# Replace the first 2 occurrences
```

```
x = re.sub("\s", ".", txt, 2)
```

```
print(x)
```

```
The.rain.in Ankara
```

```
import re
```

```
txt = "The rain in Turkey"
```

```
# Do a search
```

```
x = re.search("ai", txt)
```

```
print(x)
```

```
<re.Match object; span=(5, 7), match='ai'>
```

```
# .span() returns a tuple containing the start-, and end
```

```
x = re.search("ai", txt)
```

```
print(x.span())
```

```
(5, 7)
```

PYTHON PROGRAMMING

Regex Examples

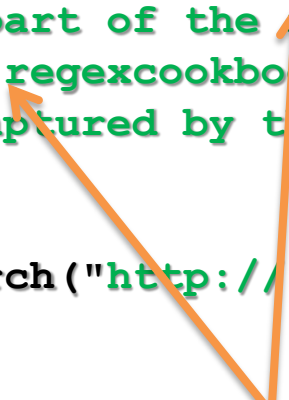
search() function

```
import re
subject = "Please visit http://www.regexcookbook.com for more
information. The part of the regex inside the first capturing
group matches www.regexcookbook.com, and you want to retrieve
the domain name captured by the first capturing group into a
string variable."

matchobj = re.search("http://([a-z0-9.-]+)", subject)

if matchobj:
    result = matchobj.group(1)
else:
    result = ""

print(result)
```



```
import re
```

```
text = 'The vote was 65 in favour, 43 against and 21  
abstentions'  
match = re.search(r'(\d+).* (\d+).* (\d+)', text)  
print(match.group(1), match.group(2), match.group(3))
```

This code will print: 65 2 1

You might have expected to see 65 43 21. The reason for this output is that the

.* regular expression is greedy, which means it will match as much as it can.

Here's what happened:

- The first `.*(\d+)` will match 65.
- The `.*` after it will match in favour, 43 against and.
- The next `.*(\d+)` will match 2.
- The `.*` after it will match the empty string since `*` means zero or more.
- The final `.*(\d+)` will match 1.