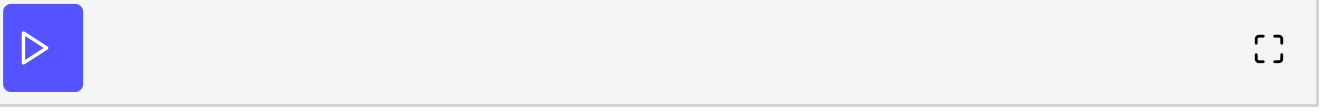
Evaluating Arbitrary Strings As Python Expressions

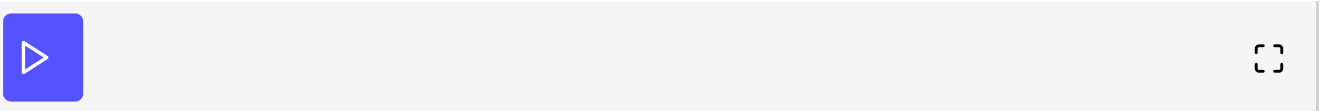
This is the final piece of the puzzle (or rather, the final piece of the puzzle solver). After all that fancy string manipulation, we're left with a string like `'9567 + 1085 == 10652'`. But that's a string, and what good is a string? Enter `eval()`, the universal Python evaluation tool.

```
print (eval('1 + 1 == 2'))
#True

print (eval('1 + 1 == 3'))
#False

print (eval('9567 + 1085 == 10652'))
#True
```



But wait, there's more! The `eval()` function isn't limited to boolean expressions. It can handle *any* Python expression and returns *any* datatype.

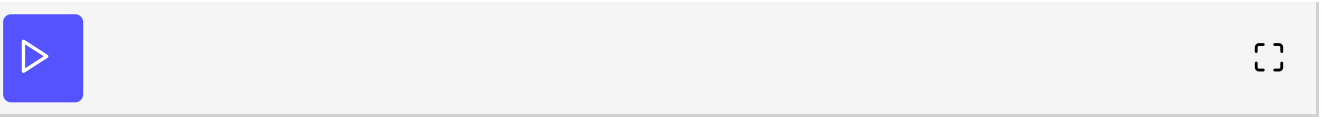
```
print (eval('"A" + "B"'))
#AB
print (eval('"MARK".translate({65: 79})'))
#MORK
print (eval('"AAAAA".count("A")'))
#5
print (eval('["*"] * 5'))
#['*', '*', '*', '*', '*']
```



But wait, that's not all!

```
x = 5
print (eval("x * 5"))         #①
#25
```

```
print (eval("pow(x, 2)"))     #②
#25


import math
print (eval("math.sqrt(x)"))  #③
#2.23606797749979
```



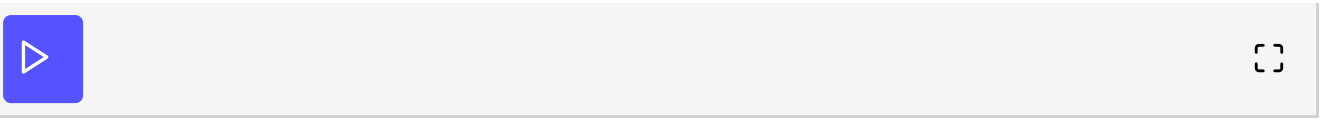
① The expression that `eval()` takes can reference global variables defined outside the `eval()`. If called within a function, it can reference local variables too.

② And functions.

③ And modules.

Hey, wait a minute…

```
import subprocess
print (eval("subprocess.getoutput('ls ~')"))                  #①
#ls: cannot access /nonexistent: No such file or directory

print (eval("subprocess.getoutput('rm /some/random/file')"))  #②
#rm: cannot remove '/some/random/file': No such file or directory
```



① The `subprocess` module allows you to run arbitrary shell commands and get the result as a Python string.

② Arbitrary shell commands can have permanent consequences.

It's even worse than that, because there's a global `__import__()` function that takes a module name as a string, imports the module, and returns a reference to it. Combined with the power of `eval()`, you can construct a single expression that will wipe out all your files:

```
eval("__import__('subprocess').getoutput('rm /some/random/file')")  #①
```

① Now imagine the output of `'rm -rf ~'`. Actually there wouldn't be any output, but you wouldn't have any files left either.

eval() is EVIL

Well, the evil part is evaluating arbitrary expressions from untrusted sources. You should only use `eval()` on trusted input. Of course, the trick is figuring out what's "trusted." But here's something I know for certain: you should **NOT** take this alphametics solver and put it on the internet as a fun little web service. Don't make the mistake of thinking, "Gosh, the function does a lot of string manipulation before getting a string to evaluate; *I can't imagine* how someone could exploit that." Someone **WILL** figure out how to sneak nasty executable code past all that string manipulation (stranger things have happened), and then you can kiss your server goodbye.

But surely there's *some* way to evaluate expressions safely? To put `eval()` in a sandbox where it can't access or harm the outside world? Well, yes and no.

```
x = 5
eval("x * 5", {}, {})                              #①
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 2, in <module>
# eval("x * 5", {}, {}) # \u2460
# File "<string>", line 1, in <module>
#NameError: name 'x' is not defined
```

```
x = 5
print (eval("x * 5", {"x": x}, {}) )               #②
#25
```

```
x = 5

import math

eval("math.sqrt(x)", {"x": x}, {})                 #③
#Traceback (most recent call last):
#   File "/usercode/__ed_file.py", line 5, in <module>
# eval("math.sqrt(x)", {"x": x}, {}) #\u2462
# File "<string>", line 1, in <module>
```

```
#NameError: name 'math' is not defined
```

① The second and third parameters passed to the `eval()` function act as the global and local namespaces for evaluating the expression. In this case, they are both empty, which means that when the string `"x * 5"` is evaluated, there is no reference to `x` in either the global or local namespace, so `eval()` throws an exception.

② You can selectively include specific values in the global namespace by listing them individually. Then those — and only those — variables will be available during evaluation.

③ Even though you just imported the `math` module, you didn't include it in the namespace passed to the `eval()` function, so the evaluation failed.

Gee, that was easy. Lemme make an alphametics web service now!

```
print (eval("pow(5, 2)", {}, {}))                  #①
#25

print (eval("__import__('math').sqrt(5)", {}, {}))  #②
#2.23606797749979
```

① Even though you've passed empty dictionaries for the global and local namespaces, all of Python's built-in functions are still available during evaluation. So `pow(5, 2)` works, because `5` and `2` are literals, and `pow()` is a built-in function.

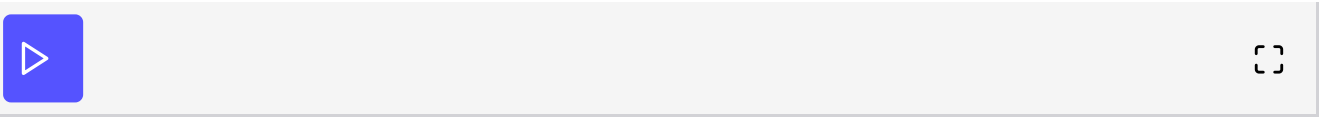
② Unfortunately (and if you don't see why it's unfortunate, read on), the `__import__()` function is also a built-in function, so it works too.

Yeah, that means you can still do nasty things, even if you explicitly set the global and local namespaces to empty dictionaries when calling `eval()`:

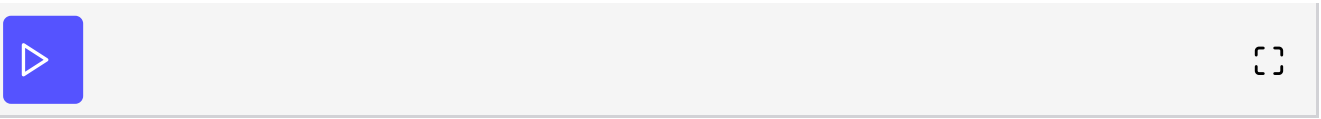
```
eval("__import__('subprocess').getoutput('rm /some/random/file')", {}, {})
```

Oops. I'm glad I didn't make that alphametics web service. Is there *any* way to use `eval()` safely? Well, yes and no.

```
eval("__import__('math').sqrt(5)",
     {"__builtins__":None}, {})          #①
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 2, in <module>
# {"__builtins__":None}, {}) #\u2460
# File "<string>", line 1, in <module>
#TypeError: 'NoneType' object is not subscriptable
```



```
eval("__import__('subprocess').getoutput('rm -rf /')",
     {"__builtins__":None}, {})          #②
# Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 2, in <module>
# {"__builtins__":None}, {}) #\u2461
# File "<string>", line 1, in <module>
#TypeError: 'NoneType' object is not subscriptable
```



① To evaluate untrusted expressions safely, you need to define a global namespace dictionary that maps `"__builtins__"` to `None`, the Python null value. Internally, the "built-in" functions are contained within a pseudo-module called `"__builtins__"`. This pseudo-module (i.e. the set of built-in functions) is made available to evaluated expressions unless you explicitly override it.

② Be sure you've overridden `__builtins__`. Not `__builtin__`, `__built-ins__`, or some other variation that will work just fine but expose you to catastrophic risks.

So `eval()` is safe now? Well, yes and no.

```
eval("2 ** 2147483647",
     {"__builtins__":None}, {})          #①
```

① Even without access to `__builtins__`, you can still launch a denial-of-service attack. For example, trying to raise `2` to the `2147483647th` power will spike your server's CPU utilization to 100% for quite some time. (If you're trying this in the interactive shell, press `Ctrl-C` a few times to break out of it.)

Technically this expression *will* return a value eventually, but in the meantime your server will be doing a whole lot of nothing.

In the end, it *is* possible to safely evaluate untrusted Python expressions, for some definition of “safe” that turns out not to be terribly useful in real life. It’s fine if you’re just playing around, and it’s fine if you only ever pass it trusted input. But anything else is just asking for trouble.