Python eval(): Evaluate Expressions Dynamically

by Leodanis Pozo Ramos 4 Comments intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Evaluate Expressions Dynamically With Python eval()**

Python's **`eval()`** allows you to evaluate arbitrary Python **expressions** from a string-based or compiled-code-based input. This function can be handy when you're trying to dynamically evaluate Python expressions from any input that comes as a string or a compiled code object.

Although Python's `eval()` is an incredibly useful tool, the function has some important security implications that you should consider before using it. In this tutorial, you'll learn how `eval()` works and how to use it safely and effectively in your Python programs.

Help

In this tutorial, you'll learn:

- How Python's **`eval()`** works
- How to use `eval()` to **dynamically evaluate** arbitrary string-based or compiled-code-based input
- How `eval()` can make your code insecure and how to minimize the associated **security risks**

Additionally, you'll learn how to use Python's `eval()` to code an application that interactively evaluates math expressions. With this example, you'll apply everything you've learned about `eval()` to a real-world problem. If you want to get the code for this application, then you can click on the box below:

> **Download the sample code: Click here to get the code you'll use** to learn about Python's eval() in this tutorial.

Understanding Python's `eval()`

You can use the built-in Python **`eval()`** to dynamically evaluate expressions from a string-based or compiled-code-based input. If you pass in a string to `eval()`, then the function parses it, compiles it to bytecode, and evaluates it as a Python expression. But if you call `eval()` with a compiled code object, then the function performs just the evaluation step, which is quite convenient if you call `eval()` several times with the same input.

The signature of Python's `eval()` is defined as follows:

Python

```
eval(expression[, globals[, locals]])
```

The function takes a first argument, called `expression`, which holds the expression that you need to evaluate. `eval()` also takes two optional arguments:

1. `globals`
2. `locals`

In the next three sections, you'll learn what these arguments are and how `eval()` uses them to evaluate Python expressions on the fly.

> **Note:** You can also use **`exec()`** to dynamically execute Python code. The main difference between `eval()` and `exec()` is that `eval()` can only execute or evaluate expressions, whereas `exec()` can execute any piece of Python code.

The First Argument: `expression`

The first argument to `eval()` is called **`expression`**. It's a required argument that holds the **string-based** or **compiled-code-based** input to the function. When you call `eval()`, the content of `expression` is evaluated as a Python expression. Check out the following examples that use string-based input:

Python >>>

```
>>> eval("2 ** 8")
256
>>> eval("1024 + 1024")
2048
>>> eval("sum([8, 16, 32])")
56
>>> x = 100
>>> eval("x * 2")
200
```

When you call `eval()` with a string as an argument, the function returns the value that results from evaluating the input string. By default, `eval()` has access to global names like `x` in the above example.

To evaluate a string-based `expression`, Python's `eval()` runs the following steps:

1. **Parse** `expression`
2. **Compile** it to bytecode
3. **Evaluate** it as a Python expression

4. **Return** the result of the evaluation

The name `expression` for the first argument to `eval()` highlights that the function works only with expressions and not with compound statements. The Python documentation defines **expression** as follows:

> **expression**
>
> A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions. (Source)

On the other hand, a Python **statement** has the following definition:

> **statement**
>
> A statement is part of a suite (a "block" of code). A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`. (Source)

If you try to pass a compound statement to `eval()`, then you'll get a `SyntaxError`. Take a look at the following example in which you try to execute an `if` statement using `eval()`:

Python >>>

```
>>> x = 100
>>> eval("if x: print(x)")
  File "<string>", line 1
    if x: print(x)
    ^
SyntaxError: invalid syntax
```

If you try to evaluate a compound statement using Python's `eval()`, then you'll get a `SyntaxError` like in the above traceback. That's because `eval()` only accepts expressions. Any other statement, such as `if`, `for`, `while`, `import`, `def`, or `class`, will raise an error.

> **Note:** A `for` loop is a compound statement, but the `for` keyword can also be used in comprehensions, which are considered expressions. You can use `eval()` to evaluate comprehensions even though they use the `for` keyword.

Assignment operations aren't allowed with `eval()` either:

Python >>>

```
>>> eval("pi = 3.1416")
  File "<string>", line 1
    pi = 3.1416
       ^
SyntaxError: invalid syntax
```

If you try to pass an assignment operation as an argument to Python's `eval()`, then you'll get a `SyntaxError`. Assignment operations are statements rather than expressions, and statements aren't allowed with `eval()`.

You'll also get a `SyntaxError` any time the parser doesn't understand the input expression. Take a look at the following example in which you try to evaluate an expression that violates Python syntax:

Python >>>

```
>>> # Incomplete expression
>>> eval("5 + 7 *")
  File "<string>", line 1
    5 + 7 *
          ^
SyntaxError: unexpected EOF while parsing
```

You can't pass an expression to `eval()` that violates Python syntax. In the above example, you try to evaluate an incomplete expression (`"5 + 7 *"`) and get a `SyntaxError` because the parser doesn't understand the syntax of the expression.

You can also pass compiled code objects to Python's `eval()`. To compile the code that you're going to pass to `eval()`, you can use `compile()`. This is a built-in function that can compile an input string into a code object or an AST object so that you can evaluate it with `eval()`.

The details of how to use `compile()` are beyond the scope of this tutorial, but here's a quick look at its first three required arguments:

1. **`source`** holds the source code that you want to compile. This argument accepts normal strings, byte strings, and AST objects.
2. **`filename`** gives the file from which the code was read. If you're going to use a string-based input, then the value for this argument should be `"<string>"`.
3. **`mode`** specifies which kind of compiled code you want to get. If you want to process the compiled code with `eval()`, then this argument should be set to `"eval"`.

> **Note:** For more information on `compile()`, check out the official documentation.

You can use `compile()` to supply code objects to `eval()` instead of normal strings. Check out the following examples:

Python >>>

```
>>> # Arithmetic operations
>>> code = compile("5 + 4", "<string>", "eval")
>>> eval(code)
9
>>> code = compile("(5 + 7) * 2", "<string>", "eval")
>>> eval(code)
24
>>> import math
>>> # Volume of a sphere
>>> code = compile("4 / 3 * math.pi * math.pow(25, 3)", "<string>", "eval")
>>> eval(code)
65449.84694978735
```

If you use `compile()` to compile the expressions that you're going to pass to `eval()`, then `eval()` goes through the following steps:

1. **Evaluate** the compiled code
2. **Return** the result of the evaluation

If you call Python's `eval()` using a compiled-code-based input, then the function performs the evaluation step and immediately returns the result. This can be handy when you need to evaluate the same expression multiple times. In this case, it's best to precompile the expression and reuse the resulting bytecode on subsequent calls to `eval()`.

If you compile the input expression beforehand, then successive calls to `eval()` will run faster because you won't be repeating the **parsing** and **compiling** steps. Unneeded repetitions can lead to high CPU times and excessive memory consumption if you're evaluating complex expressions.

The Second Argument: `globals`

The second argument to `eval()` is called **`globals`**. It's optional and holds a dictionary that provides a global namespace to `eval()`. With `globals`, you can tell `eval()` which global names to use while evaluating `expression`.

Global names are all those names that are available in your current global scope or namespace. You can access them from anywhere in your code.

All the names passed to `globals` in a dictionary will be available to `eval()` at execution time. Check out the following example, which shows how to use a custom dictionary to supply a global namespace to `eval()`:

Python >>>

```
>>> x = 100  # A global variable
>>> eval("x + 100", {"x": x})
200
>>> y = 200  # Another global variable
>>> eval("x + y", {"x": x})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'y' is not defined
```

If you supply a custom dictionary to the `globals` argument of `eval()`, then `eval()` will take only those names as globals. Any global names defined outside of this custom dictionary won't be accessible from inside `eval()`. That's why Python raises a `NameError` when you try to access `y` in the above code: The dictionary passed to `globals` doesn't include `y`.

You can insert names into `globals` by listing them in your dictionary, and then those names will be available during the evaluation process. For example, if you insert `y` into `globals`, then the evaluation of `"x + y"` in the above example will work as expected:

Python >>>

```
>>> eval("x + y", {"x": x, "y": y})
300
```

Since you add `y` to your custom `globals` dictionary, the evaluation of `"x + y"` is successful and you get the expected return value of `300`.

You can also supply names that don't exist in your current global scope. For this to work, you need to supply a concrete value for each name. `eval()` will interpret these names as global names when running:

Python >>>

```
>>> eval("x + y + z", {"x": x, "y": y, "z": 300})
600
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

Even though `z` isn't defined in your current global scope, the variable is present in `globals` with a value of `300`. In this case, `eval()` has access to `z` as though it were a global variable.

The mechanism behind `globals` is quite flexible. You can pass any visible variable (global, local, or nonlocal) to `globals`. You can also pass custom key-value pairs like `"z": 300` in the above example. `eval()` will treat all of them as global variables.

An important point regarding `globals` is that if you supply a custom dictionary to it that doesn't contain a value for the key `"__builtins__"`, then a reference to the dictionary of `builtins` will be automatically inserted under `"__builtins__"` before `expression` gets parsed. This ensures that `eval()` will have full access to all of Python's built-in names when evaluating `expression`.

The following examples show that even though you supply an empty dictionary to `globals`, the call to `eval()` will still have access to Python's built-in names:

Python >>>

```
>>> eval("sum([2, 2, 2])", {})
6
>>> eval("min([1, 2, 3])", {})
1
>>> eval("pow(10, 2)", {})
100
```

In the above code, you supply an empty dictionary (`{}`) to `globals`. Since that dictionary doesn't contain a key called `"__builtins__"`, Python automatically inserts one with a reference to the names in `builtins`. This way, `eval()` has full access to all of Python's built-in names when it parses `expression`.

If you call `eval()` without passing a custom dictionary to `globals`, then the argument will default to the dictionary returned by `globals()` in the environment where `eval()` is called:

Python >>>

```
>>> x = 100  # A global variable
>>> y = 200  # Another global variable
>>> eval("x + y")  # Access both global variables
300
```

When you call `eval()` without supplying a `globals` argument, the function evaluates `expression` using the dictionary returned by `globals()` as its global namespace. So, in the above example, you can freely access `x` and `y` because they're global variables included in your current global scope.

The Third Argument: `locals`

Python's `eval()` takes a third argument called **`locals`**. This is another optional argument that holds a dictionary. In this case, the dictionary contains the variables that `eval()` uses as local names when evaluating `expression`.

Local names are those names (variables, functions, classes, and so on) that you define inside a given function. Local names are visible only from inside the enclosing function. You define these kinds of names when you're writing a function.

Since `eval()` is already written, you can't add local names to its code or local scope. However, you can pass a dictionary to `locals`, and `eval()` will treat those names as local names:

Python >>>

```
>>> eval("x + 100", {}, {"x": 100})
200
>>> eval("x + y", {}, {"x": 100})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'y' is not defined
```

The second dictionary in the first call to `eval()` holds the variable `x`. This variable is interpreted by `eval()` as a local variable. In other words, it's seen as a variable defined in the body of `eval()`.

You can use `x` in `expression`, and `eval()` will have access to it. In contrast, if you try to use `y`, then you'll get a `NameError` because `y` isn't defined in either the `globals` namespace or the `locals` namespace.

Like with `globals`, you can pass any visible variable (global, local, or nonlocal) to `locals`. You can also pass custom key-value pairs like `"x": 100` in the above example. `eval()` will treat all of them as local variables.

Note that to supply a dictionary to `locals`, you first need to supply a dictionary to `globals`. It's not possible to use keyword arguments with `eval()`:

Python >>>

```
>>> eval("x + 100", locals={"x": 100})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: eval() takes no keyword arguments
```

If you try to use keyword arguments when calling `eval()`, then you'll get a `TypeError` explaining that `eval()` takes no keyword arguments. So, you need to supply a `globals` dictionary before you can supply a `locals` dictionary.

If you don't pass a dictionary to `locals`, then it defaults to the dictionary passed to `globals`. Here's an example in which you pass an empty dictionary to `globals` and nothing to `locals`:

Python >>>

```
>>> x = 100
>>> eval("x + 100", {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
```

Given that you don't provide a custom dictionary to `locals`, the argument defaults to the dictionary passed to `globals`. In this case, `eval()` doesn't have access to `x` because `globals` holds an empty dictionary.

The main practical difference between `globals` and `locals` is that Python will automatically insert a `"__builtins__"` key into `globals` if that key doesn't already exist. This happens whether or not you supply a custom dictionary to `globals`. On the other hand, if you supply a custom dictionary to `locals`, then that dictionary will remain unchanged during the execution of `eval()`.

Evaluating Expressions With Python's `eval()`

You can use Python's `eval()` to evaluate any kind of Python expression but not Python statements such as keyword-based compound statements or assignment statements.

`eval()` can be handy when you need to dynamically evaluate expressions and using other Python techniques or tools would considerably increase your development time and effort. In this section, you'll learn how you can use Python's `eval()` to evaluate Boolean, math, and general-purpose Python expressions.

Boolean Expressions

Boolean expressions are Python expressions that return a truth value (`True` or `False`) when the interpreter evaluates them. They're commonly used in `if` statements to check if some condition is true or false. Since Boolean expressions aren't compound statements, you can use `eval()` to evaluate them:

Python >>>

```
>>> x = 100
>>> y = 100
>>> eval("x != y")
False
>>> eval("x < 200 and y > 100")
False
>>> eval("x is y")
True
>>> eval("x in {50, 100, 150, 200}")
True
```

You can use `eval()` with Boolean expressions that use any of the following Python operators:

- **Value comparison operators**: <, >, <=, >=, ==, `!=`
- **Logical (Boolean) operators**: `and`, `or`, `not`
- **Membership test operators**: `in`, `not in`
- **Identity operators**: `is`, `is not`

In all cases, the function returns the truth value of the expression that you're evaluating.

Now, you may be thinking, why should I use `eval()` instead of using the Boolean expression directly? Well, suppose you need to implement a conditional statement, but you want to change the condition on the fly:

Python >>>

```
>>> def func(a, b, condition):
...     if eval(condition):
...         return a + b
...     return a - b
...
>>> func(2, 4, "a > b")
-2
>>> func(2, 4, "a < b")
6
>>> func(2, 2, "a is b")
4
```

Inside `func()`, you use `eval()` to evaluate the supplied `condition` and return either `a + b` or `a - b` according to the result of the evaluation. You use just a few different conditions in the above example, but you could use any number of others provided that you stick with the names `a` and `b` that you defined in `func()`.

Now imagine how you would implement something like this without the use of Python's `eval()`. Would that take less code and time? No way!

Math Expressions

One common use case of Python's `eval()` is to evaluate math expressions from a string-based input. For example, if you want to create a Python calculator, then you can use `eval()` to evaluate the user's input and return the result of the calculations.

The following examples show how you can use `eval()` along with `math` to perform math operations:

Python >>>

```
>>> # Arithmetic operations
>>> eval("5 + 7")
12
>>> eval("5 * 7")
35
>>> eval("5 ** 7")
78125
>>> eval("(5 + 7) / 2")
6.0
>>> import math
>>> # Area of a circle
>>> eval("math.pi * pow(25, 2)")
1963.4954084936207
>>> # Volume of a sphere
>>> eval("4 / 3 * math.pi * math.pow(25, 3)")
65449.84694978735
>>> # Hypotenuse of a right triangle
>>> eval("math.sqrt(math.pow(10, 2) + math.pow(15, 2))")
18.027756377319946
```

When you use `eval()` to evaluate math expressions, you can pass in expressions of any kind or complexity. `eval()` will parse them, evaluate them and, if everything is okay, give you the expected result.

General-Purpose Expressions

So far, you've learned how to use `eval()` with Boolean and math expressions. However, you can use `eval()` with more complex Python expressions that incorporate function calls, object creation, attribute access, comprehensions, and so on.

For example, you can call a built-in function or one that you've imported with a standard or third-party module:

Python >>>

```
>>> # Run the echo command
>>> import subprocess
>>> eval("subprocess.getoutput('echo Hello, World')")
'Hello, World'
>>> # Launch Firefox (if available)
>>> eval("subprocess.getoutput('firefox')")
''
```

In this example, you use Python's `eval()` to execute a few system commands. As you can imagine, you can do a *ton* of useful things with this feature. However, `eval()` can also expose you to serious security risks, like allowing a malicious user to run system commands or any arbitrary piece of code in your machine.

In the next section, you'll look at ways to address some of the security risks associated with eval().

Minimizing the Security Issues of `eval()`

Although it has an almost unlimited number of uses, Python's `eval()` also has important **security implications**. `eval()` is considered insecure because it allows you (or your users) to dynamically execute arbitrary Python code.

This is considered bad programming practice because the code that you're reading (or writing) is *not* the code that you'll execute. If you're planning to use `eval()` to evaluate input from a user or any other external source, then you won't know for sure what code is going to be executed. That's a serious security risk if your application runs in the wrong hands.

For this reason, good programming practices generally recommend against using `eval()`. But if you choose to use the function anyway, then the rule of thumb is to *never ever* use it with **untrusted input**. The tricky part of this rule is figuring out which kinds of input you can trust.

As an example of how using `eval()` irresponsibly can make your code insecure, suppose you want to build an online service for evaluating arbitrary Python expressions. Your user will introduce expressions and then click the `Run` button. The application will get the user's input and pass it to `eval()` for evaluation.

This application will run on your personal server. Yes, the same server where you have all those valuable files. If you're running a Linux box and the application's process has the right permissions, then a malicious user could introduce a dangerous string like the following:

Python

```
"__import__('subprocess').getoutput('rm -rf *')"
```

The above code would delete all the files in the application's current directory. That would be awful, wouldn't it?

> **Note:** `__import__()` is a built-in function that takes a module name as a string and returns a reference to the module object. `__import__()` is a function, which is totally different from an `import` statement. You can't evaluate an `import` statement using `eval()`.

When the input is untrusted, there's no completely effective way to avoid the security risks associated with `eval()`. However, you can minimize your risk by restricting the execution environment of `eval()`. You'll learn a few techniques for doing so in the following sections.

Restricting `globals` and `locals`

You can restrict the execution environment of `eval()` by passing custom dictionaries to the `globals` and `locals` arguments. For example, you can pass empty dictionaries to both arguments to prevent `eval()` from accessing names in the caller's current scope or namespace:

Python >>>

```
>>> # Avoid access to names in the caller's current scope
>>> x = 100
>>> eval("x * 5", {}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
```

If you pass empty dictionaries (`{}`) to `globals` and `locals`, then `eval()` won't find the name `x` in either its global namespace or its local namespace when evaluating the string `"x * 5"`. As a result, `eval()` will throw a `NameError`.

Unfortunately, restricting the `globals` and `locals` arguments like this doesn't eliminate all the security risks associated with the use of Python's `eval()`, because you can still access all of Python's built-in names.

Restricting the Use of Built-In Names

As you saw earlier, Python's `eval()` automatically inserts a reference to the dictionary of `builtins` into `globals` before parsing `expression`. A malicious user could exploit this behavior by using the built-in function `__import__()` to get access to the standard library and any third-party module that you've installed on your system.

The following examples show that you can use any built-in function and any standard module like `math` or `subprocess` even after you've restricted `globals` and `locals`:

Python >>>

```
>>> eval("sum([5, 5, 5])", {}, {})
15
>>> eval("__import__('math').sqrt(25)", {}, {})
5.0
>>> eval("__import__('subprocess').getoutput('echo Hello, World')", {}, {})
'Hello, World'
```

Even though you restrict `globals` and `locals` using empty dictionaries, you can still use any built-in function like you did with `sum()` and `__import__()` in the above code.

You can use `__import__()` to import any standard or third-party module just like you did above with `math` and `subprocess`. With this technique, you can access any function or class defined in `math`, `subprocess`, or any other module. Now imagine what a malicious user could do to your system using `subprocess` or any other powerful module in the standard library.

To minimize this risk, you can restrict access to Python's built-in functions by overriding the `"__builtins__"` key in `globals`. Good practice recommends using a custom dictionary containing the key-value pair `"__builtins__": {}`. Check out the following example:

Python >>>

```
>>> eval("__import__('math').sqrt(25)", {"__builtins__": {}}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

If you pass a dictionary containing the key-value pair `"__builtins__": {}` to `globals`, then `eval()` won't have direct access to Python's built-in functions like `__import__()`. However, as you'll see in the next section, this approach still doesn't make `eval()` completely secure.

Restricting Names in the Input

Even though you can restrict the execution environment of Python's `eval()` using custom `globals` and `locals` dictionaries, the function will still be vulnerable to a few fancy tricks. For example, you can access the class `object` using a **type literal** like `""`, `[]`, `{}`, or `()` along with some special attributes:

Python >>>

```
>>> "".__class__.__base__
<class 'object'>
>>> [].__class__.__base__
<class 'object'>
>>> {}.__class__.__base__
<class 'object'>
>>> ().__class__.__base__
<class 'object'>
```

Once you have access to `object`, you can use the special method `.__subclasses__()` to get access to all of the classes that inherit from `object`. Here's how it works:

Python >>>

```
>>> for sub_class in ().__class__.__base__.__subclasses__():
...     print(sub_class.__name__)
...
type
weakref
weakcallableproxy
weakproxy
int
...
```

This code will print a large list of classes to your screen. Some of these classes are quite powerful and can be extremely dangerous in the wrong hands. This opens up another important security hole that you can't close by simply restricting the execution environment of `eval()`:

Python >>>

```
>>> input_string = """[
...     c for c in ().__class__.__base__.__subclasses__()
...     if c.__name__ == "range"
... ][0](10)"""
>>> list(eval(input_string, {"__builtins__": {}}, {}))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The list comprehension in the above code filters the classes that inherit from `object` to return a `list` containing the class `range`. The first index (`[0]`) returns the class `range`. Once you have access to `range`, you call it to generate a `range` object. Then you call `list()` on the `range` object to generate a list of ten integers.

In this example, you use `range` to illustrate a security hole in `eval()`. Now imagine what a malicious user could do if your system exposed classes like `subprocess.Popen`.

> **Note:** For a deeper dive into the vulnerabilities of `eval()`, check out Ned Batchelder's article, Eval really is dangerous.

A possible solution to this vulnerability is to restrict the use of names in the input, either to a bunch of *safe* names or to *no* names at all. To implement this technique, you need to go through the following steps:

1. **Create** a dictionary containing the names that you want to use with `eval()`.
2. **Compile** the input string to bytecode using `compile()` in mode `"eval"`.
3. **Check** `.co_names` on the bytecode object to make sure it contains only allowed names.
4. **Raise** a `NameError` if the user tries to enter a name that's not allowed.

Take a look at the following function in which you implement all these steps:

Python >>>

```
>>> def eval_expression(input_string):
...     # Step 1
...     allowed_names = {"sum": sum}
...     # Step 2
...     code = compile(input_string, "<string>", "eval")
...     # Step 3
...     for name in code.co_names:
...         if name not in allowed_names:
...             # Step 4
...             raise NameError(f"Use of {name} not allowed")
...     return eval(code, {"__builtins__": {}}, allowed_names)
```

In `eval_expression()`, you implement all of the steps you saw before. This function restricts the names that you can use with `eval()` to only those names in the dictionary `allowed_names`. To do this, the function uses `.co_names`, which is an attribute of a code object that returns a tuple containing the names in the code object.

The following examples show how `eval_expression()` works in practice:

Python >>>

```
>>> eval_expression("3 + 4 * 5 + 25 / 2")
35.5
>>> eval_expression("sum([1, 2, 3])")
6
>>> eval_expression("len([1, 2, 3])")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in eval_expression
NameError: Use of len not allowed
>>> eval_expression("pow(10, 2)")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in eval_expression
NameError: Use of pow not allowed
```

If you call `eval_expression()` to evaluate arithmetic operations, or if you use expressions that include allowed names, then you'll get the expected result. Otherwise, you'll get a `NameError`. In the above examples, the only name you've allowed is `sum()`. Other names like `len()` and `pow()` are not allowed, so the function raises a `NameError` when you try to use them.

If you want to completely disallow the use of names, then you can rewrite `eval_expression()` as follows:

Python >>>

```
>>> def eval_expression(input_string):
...     code = compile(input_string, "<string>", "eval")
...     if code.co_names:
...         raise NameError(f"Use of names not allowed")
...     return eval(code, {"__builtins__": {}}, {})
...
>>> eval_expression("3 + 4 * 5 + 25 / 2")
35.5
>>> eval_expression("sum([1, 2, 3])")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in eval_expression
NameError: Use of names not allowed
```

Now your function doesn't allow *any* names in the input string. To accomplish this, you check for names in `.co_names` and raise a `NameError` if one is found. Otherwise, you evaluate `input_string` and return the result of the evaluation. In this case, you use an empty dictionary to restrict `locals` as well.

You can use this technique to minimize the security issues of `eval()` and strengthen your armor against malicious attacks.

Restricting the Input to Only Literals

A common use case for Python's `eval()` is to evaluate strings that contain standard Python literals and turn them into concrete objects.

The standard library provides a function called `literal_eval()` that can help achieve this goal. The function doesn't support operators, but it does support lists, tuples, numbers, strings, and so on:

Python >>>

```
>>> from ast import literal_eval
>>> # Evaluating literals
>>> literal_eval("15.02")
15.02
>>> literal_eval("[1, 15]")
[1, 15]
>>> literal_eval("(1, 15)")
(1, 15)
>>> literal_eval("{'one': 1, 'two': 2}")
{'one': 1, 'two': 2}
>>> # Trying to evaluate an expression
>>> literal_eval("sum([1, 15]) + 5 + 8 * 2")
Traceback (most recent call last):
  ...
ValueError: malformed node or string: <_ast.BinOp object at 0x7faedecd7668>
```

Notice that `literal_eval()` only works with standard type literals. It doesn't support the use of operators or names. If you try to feed an expression to `literal_eval()`, then you'll get a `ValueError`. This function can also help you minimize the security risks associated with the use of Python's `eval()`.

Using Python's `eval()` With `input()`

In Python 3.x, the built-in **`input()`** reads the user input at the command line, converts it to a string, strips the trailing newline, and returns the result to the caller. Since the result of `input()` is a string, you can feed it to `eval()` and evaluate it as a Python expression:

Python >>>

```
>>> eval(input("Enter a math expression: "))
Enter a math expression: 15 * 2
30
>>> eval(input("Enter a math expression: "))
Enter a math expression: 5 + 8
13
```

You can wrap Python's `eval()` around `input()` to automatically evaluate the user's input. This is a common use case for `eval()` because it emulates the behavior of `input()` in Python 2.x, in which `input()` evaluates the user's input as a Python expression and returns the result.

This behavior of `input()` in Python 2.x was changed in Python 3.x because of its security implications.

Building a Math Expressions Evaluator

So far, you've learned how Python's `eval()` works and how to use it in practice. You've also learned that `eval()` has important security implications and that it's generally considered good practice to avoid the use of `eval()` in your code. However, there are some situations in which Python's `eval()` can save you a lot of time and effort.

In this section, you're going to code an application to evaluate math expressions on the fly. If you wanted to solve this problem without using `eval()`, then you'd need to go through the following steps:

1. **Parse** the input expression.
2. **Change** the expression's components into Python objects (numbers, operators, functions, and so on).
3. **Combine** everything into an expression.
4. **Confirm** that the expression is valid in Python.
5. **Evaluate** the final expression and return the result.

That would be a lot of work considering the wide variety of possible expressions that Python can process and evaluate. Fortunately, you can use `eval()` to solve this problem, and you've already learned several techniques to reduce the associated security risks.

You can get the source code for the application that you're going to build in this section by clicking on the box below:

Download the sample code: Click here to get the code you'll use to learn about Python's eval() in this tutorial.

First, fire up your favorite code editor. Create a new Python script called `mathrepl.py`, and then add the following code:

Python

```
1  import math
2
3  __version__ = "1.0"
4
5  ALLOWED_NAMES = {
6      k: v for k, v in math.__dict__.items() if not k.startswith("__")
7  }
8
9  PS1 = "mr>>"
10
11 WELCOME = f"""
12 MathREPL {__version__}, your Python math expressions evaluator!
13 Enter a valid math expression after the prompt "{PS1}".
14 Type "help" for more information.
15 Type "quit" or "exit" to exit.
16 """
17
18 USAGE = f"""
19 Usage:
20 Build math expressions using numeric values and operators.
21 Use any of the following functions and constants:
22
23 {', '.join(ALLOWED_NAMES.keys())}
24 """
```

In this piece of code, you first import Python's `math` module. This module will allow you to perform math operations using predefined functions and constants. The constant `ALLOWED_NAMES` holds a dictionary containing the non-special names in `math`. This way, you'll be able to use them with `eval()`.

You also define three more string constants. You'll use them as the user interface to your script and you'll print them to the screen as needed.

Now you're ready to code the core functionality of your application. In this case, you want to code a function that receives math expressions as input and returns their result. To do this, you write a function called `evaluate()`:

Python

```
26 def evaluate(expression):
27     """Evaluate a math expression."""
28     # Compile the expression
29     code = compile(expression, "<string>", "eval")
30
31     # Validate allowed names
32     for name in code.co_names:
33         if name not in ALLOWED_NAMES:
34             raise NameError(f"The use of '{name}' is not allowed")
35
36     return eval(code, {"__builtins__": {}}, ALLOWED_NAMES)
```

Here's how the function works:

1. In **line 26**, you define `evaluate()`. This function takes the string `expression` as an argument and returns a float that represents the result of evaluating the string as a math expression.

2. In **line 29**, you use `compile()` to turn the input string `expression` into compiled Python code. The compiling operation will raise a `SyntaxError` if the user enters an invalid expression.

3. In **line 32**, you start a `for` loop to inspect the names contained in `expression` and confirm that they can be used in the final expression. If the user provides a name that is not in the list of allowed names, then you raise a `NameError`.

4. In **line 36**, you perform the actual evaluation of the math expression. Notice that you pass custom dictionaries to `globals` and `locals` as good practice recommends. `ALLOWED_NAMES` holds the functions and constants defined in `math`.

> **Note:** Since this application uses the functions defined in `math`, you need to consider that some of these functions will raise a `ValueError` when you call them with an invalid input value.
>
> For example, `math.sqrt(-10)` would raise an error because the square root of `-10` is undefined. Later on, you'll see how to catch this error in your client code.

The use of custom values for the `globals` and `locals` parameters, along with the check of names in **line 33**, allows you to minimize the security risks associated with the use of `eval()`.

Your math expression evaluator will be finished when you write its client code in `main()`. In this function, you'll define the program's main loop and close the cycle of reading and evaluating the expressions that your user enters in the command line.

For this example, the application will:

1. **Print** a welcome message to the user
2. **Show** a prompt ready to read the user's input
3. **Provide** options to get usage instructions and to terminate the application
4. **Read** the user's math expression
5. **Evaluate** the user's math expression
6. **Print** the result of the evaluation to the screen

Check out the following implementation of `main()`:

Python

```
def main():
    """Main loop: Read and evaluate user's input."""
    print(WELCOME)
    while True:
        # Read user's input
        try:
            expression = input(f"{PS1} ")
        except (KeyboardInterrupt, EOFError):
            raise SystemExit()

        # Handle special commands
        if expression.lower() == "help":
            print(USAGE)
            continue
        if expression.lower() in {"quit", "exit"}:
            raise SystemExit()

        # Evaluate the expression and handle errors
        try:
            result = evaluate(expression)
        except SyntaxError:
            # If the user enters an invalid expression
            print("Invalid input expression syntax")
            continue
        except (NameError, ValueError) as err:
            # If the user tries to use a name that isn't allowed
            # or an invalid value for a given math function
            print(err)
            continue

        # Print the result if no error occurs
        print(f"The result is: {result}")

if __name__ == "__main__":
    main()
```

Inside `main()`, you first print the `WELCOME` message. Then you read the user's input in a `try` statement to catch `KeyboardInterrupt` and `EOFError`. If either of these exceptions occur, then you terminate the application.

If the user enters the `help` option, then the application shows your `USAGE` guide. Likewise, if the user enters `quit` or `exit`, then the application terminates.

Finally, you use `evaluate()` to evaluate the user's math expression, and then you print the result to the screen. It's important to note that a call to `evaluate()` can raise the following exceptions:

- **`SyntaxError`**: This happens when the user enters an expression that doesn't follow Python syntax.
- **`NameError`**: This happens when the user tries to use a name (function, class, or attribute) that isn't allowed.
- **`ValueError`**: This happens when the user tries to use a value that isn't allowed as an input to a given function in `math`.

Notice that in `main()`, you catch all of these exceptions and print messages to the user accordingly. This will allow the user to review the expression, fix the problem, and run the program again.

That's it! You've built a math expression evaluator in about seventy lines of code using Python's `eval()`. To run the application, open your system's command line and type the following command:

Shell

```
$ python3 mathrepl.py
```

This command will launch the math expression evaluator's command-line interface (CLI). You'll see something like this on your screen:

Shell

```
MathREPL 1.0, your Python math expressions evaluator!
Enter a valid math expression after the prompt "mr>>".
Type "help" for more information.
Type "quit" or "exit" to exit.

mr>>
```

Once you're there, you can enter and evaluate any math expression. For example, type the following expressions:

Shell

```
mr>> 25 * 2
The result is: 50
mr>> sqrt(25)
The result is: 5.0
mr>> pi
The result is: 3.141592653589793
```

If you enter a valid math expression, then the application evaluates it and prints the result to your screen. If there are any problems with your expressions, then the application will tell you:

Shell

```
mr>> 5 * (25 + 4
Invalid input expression syntax
mr>> sum([1, 2, 3, 4, 5])
The use of 'sum' is not allowed
mr>> sqrt(-15)
math domain error
mr>> factorial(-15)
factorial() not defined for negative values
```

In the first example, you miss the closing parentheses, so you get a message telling you that the syntax is incorrect. Then you call `sum()`, which isn't allowed, and you get an explanatory error message. Finally, you call a `math` function with an invalid input value, and the application generates a message identifying the problem in your input.

There you have it—your math expressions evaluator is ready! Feel free to add some extra features. A few ideas to get you started include enlarging the dictionary of allowed names and adding more elaborate warning messages. Give it a shot and let us know in the comments how it goes.

Conclusion

You can use Python's **`eval()`** to evaluate Python **expressions** from a string-based or code-based input. This built-in function can be useful when you're trying to evaluate Python expressions on the fly and you want to avoid the hassle of creating your own expressions evaluator from scratch.

In this tutorial, you've learned how `eval()` works and how to use it safely and effectively to evaluate arbitrary Python expressions.

You're now able to:

- Use Python's `eval()` to dynamically **evaluate** basic Python expressions
- Run more complex statements like **function calls**, **object creation**, and **attribute access** using `eval()`
- Minimize the **security risks** associated with the use of Python's `eval()`

Additionally, you've coded an application that uses `eval()` to interactively evaluate math expressions using a command-line interface. You can download the application's code by clicking on the link below:

Download the sample code: Click here to get the code you'll use to learn about Python's eval() in this tutorial.

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the