Dynamic code execution with `exec` and `eval` All Versions

Examples

Executing code provided by untrusted user using exec, eval, or ast.literal_eval

It is not possible to use `eval` or `exec` to execute code from untrusted user securely. Even `ast.literal_eval` is prone to crashes in the parser. It is sometimes possible to guard against malicious code execution, but it doesn't exclude the possibility of outright crashes in the parser or the tokenizer.

To evaluate code by an untrusted user you need to turn to some third-party module, or perhaps write your own parser and your own virtual machine in Python.

Evaluating a string containing a Python literal with ast.literal_eval

If you have a string that contains Python literals, such as strings, floats etc, you can use `ast.literal_eval` to evaluate its value instead of `eval` . This has the added feature of allowing only certain syntax.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

However, this is not secure for execution of code provided by untrusted user, and it is trivial to crash an interpreter with carefully crafted input

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Here, the input is a string of `()` repeated one million times, which causes a crash in CPython parser. CPython developers do not consider bugs in parser as security issues.

Evaluating an expression with eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Evaluating an expression with eval using custom globals

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

As a plus, with this the code cannot accidentally refer to the names defined outside:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'variables' is not defined
```

Using `defaultdict` allows for example having undefined variables set to zero:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Evaluating statements with exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Precompiling an expression to evaluate it multiple times

`compile` built-in function can be used to precompile an expression to a code object; this code object can then be passed to eval. This will speed up the repeated executions of the evaluated code. The 3rd parameter to `compile` needs to be the string `'eval'` .

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

Syntax

```
eval(expression[, globals=None[, locals=None]])
```

```
exec(object)
```

```
exec(object, globals)
```

```
exec(object, globals, locals)
```

Parameters

Argument	Details
`expression`	The expression code as a string, or a `code` object
`object`	The statement code as a string, or a `code` object
`globals`	The dictionary to use for global variables. If `locals` is not specified, this is also used for locals. If omitted, the `globals()` of calling scope are used.
`locals`	A *mapping* object that is used for local variables. If omitted, the one passed for `globals` is used instead. If both are omitted, then the `globals()` and `locals()` of the calling scope are used for `globals` and `locals` respectively.

Remarks

In `exec` , if `globals` is `locals` (i.e. they refer to the same object), the code is executed as if it is on the module level. If `globals` and `locals` are distinct objects, the code is executed as if it were in a *class body* .

If the `globals` object is passed in, but doesn't specify `__builtins__` key, then Python built-in functions and names are automatically added to the global scope. To suppress the availability of functions such as `print` or `isinstance` in the executed scope, let `globals` have the key `__builtins__` mapped to value `None` . However, this is not a security feature.

The Python 2 -specific syntax shouldn't be used; the Python 3 syntax will work in Python 2. Thus the following forms are deprecated: <s>

- `exec object`
- `exec object in globals`
- `exec object in globals, locals`